Understanding planning with incomplete information and sensing

Marcelo Oglietti a,b

a Università di Roma “La Sapienza”, Dipartimento di Informatica e Sistemistica,
Via Salaria 113, 00198 Rome, Italy
b CONAE—Argentine National Space Agency, Paseo Colon 751, C1063ACH Buenos Aires, Argentina

Received 24 October 2004; accepted 29 November 2004
Available online 6 January 2005

Abstract

PKS is the framework for planning with incomplete information and sensing recently introduced by Bacchus and Petrick [Proc. KR’98, pp. 432–443]. The fact that PKS generalizes STRIPS to domains with incomplete information and sensing opens up the possibility of proposing it as a reference for comparisons with other formalisms that approach the problem from different perspectives.

To this end we first provide a formal semantics for PKS, then analyze and extend it. The formal definition of the extended PKS entails the identification of a number of properties of this planning framework. In particular, we prove that for any finite instance of the PKS planning problem the reachable states are finite; on the basis of this result we propose an improved planning algorithm that is not only sound, as the one proposed by Petrick and Bacchus [Proc. AIPS’02, pp. 212–221], but also complete.

We extend PKS to include conditional plans with cycles and introduce the distinction between different classes of solutions: strong, strong cyclic, weak acyclic and weak cyclic. In contrast with current belief, we prove that some weak acyclic solutions are more likely to succeed for a limited execution than some strong cyclic solutions, revealing the lack of a method for judging the quality of different solutions. Finally, we introduce a quality measure for solutions of any class, and a quantitative method for comparing them.

© 2004 Published by Elsevier B.V.

Keywords: Planning; Planning with incomplete information and sensing; Plan quality; Quality measures; Epistemic planning; Conditional planning

E-mail address: oglietti@dis.uniroma1.it (M. Oglietti).

0004-3702/S – see front matter © 2004 Published by Elsevier B.V.
1. Introduction

In AI Planning, increasing attention has been devoted to the problem of planning with incomplete information and sensing, as a response to its relevance for achieving realistic applications. Various frameworks for this kind of planning problem have been presented in recent years, e.g., [3,4,6–8,11,12,24,25].

The present paper studies one of these frameworks, namely PKS (Planning with Knowledge and Sensing), proposed by Bacchus and Petrick in [3,24]. In [3] the authors define a modeling framework for incomplete information and sensing. In [24] they introduce a planning algorithm, called PlanPKS, which produces conditional plans for incomplete information and sensing problems. We provide a formal semantics for PKS and derive some properties that are grounded on this semantics.

PKS is a generalization of STRIPS [16,21] that addresses planning with incomplete information and sensing. While STRIPS reasons on the “state of the world”, PKS reasons on the “knowledge of the agent about the world”. In fact, PKS is very much like STRIPS. This makes it of great interest because STRIPS is the basic reference language for AI planning. It is the planning framework most studied in theoretical papers; it is also the basis for the languages used in the international planning competition, indeed, for PDDL [22,23] and its extension to temporal domains PDDL 2.1 [17]. Hence, it is an important framework to be analyzed in detail.

Moreover, we consider PKS of much interest because the general familiarity with STRIPS makes PKS easier to understand and compare with previous planning formalisms than any other framework for incomplete information and sensing. In addition, various frameworks for this kind of planning problems have grown completely isolated from one another in recent years, except for the implicit comparison that relies on experimental testing against common benchmarks. Therefore, PKS can be used as a common base to understand the scope and relationship between all these planning frameworks. Certainly, to use PKS for this purpose, we need to have its semantics formally defined in detail. We define this formal semantics and then extend PKS to include conditional plans that are graphs and therefore can have cycles, as well as a distinction between classes of solutions: strong, strong cyclic, weak acyclic and weak cyclic. These extensions are necessary to perform a theoretical comparison of PKS with other frameworks.

Plan of the paper. The paper consists of this introduction and four sections.

In Section 2 we present the formal semantics for PKS. We first define what in PKS takes the place of states in STRIPS, the database-states. We also define the knowledge-states, and how these two kinds of states are related. Afterwards, we define an instance of the PKS planning problem and then what is a solution for this problem. The extended version of PKS we present deals with four kinds of solutions: strong, strong cyclic, weak acyclic and weak cyclic.

In Section 3 some results are derived, based on this formal semantics. The main outcomes are summarized as follows:

- We prove that, given a finite instance of the PKS planning problem, the reachable state space associated with that instance is finite. By reachable state space we mean the sub-
space of the database-state space that can be visited by the reasoning mechanism of PKS when searching for a solution to that instance. This result is somehow surprising in a framework with operators whose effects can add formulas with universally quantified variables.

- We describe a new planning algorithm PlanPKS* which is sound and complete for the formal semantics given here. This is an improvement with respect to the planning algorithm proposed in [24], which is sound but not complete.

- We consider complexity aspects, prove that the PKS plan existence problem is at least in $\mathcal{EA^P}$. This is caused by the presence of the operators’ parameters. The problem restricted to parameterless operators is in PSPACE.

In Section 4 we explain how to compare the quality of the different classes of solutions: strong, strong cyclic, weak acyclic, and weak cyclic. First we show that, contrary to current belief, some weak acyclic solutions have a larger probability of success after executing a finite number of steps than some strong cyclic solutions, thus, highlighting the lack of a proper method for judging the quality of different solutions. Afterwards, we advance in resolving this problem by providing a quantitative measure of the quality of solutions of any class, and a quantitative method for comparing these solutions.

The paper ends with some conclusions and a short discussion on how the proposed semantics of PKS could foster further analysis of the connections between this formalism and others that address the same problem of planning with incomplete information and sensing.

2. A formalization of PKS

Before starting our analysis of PKS, we introduce some notation used throughout the paper.

2.1. Notation

Let $\mathcal{L}$ denote a first-order language defined as usual using a tuple $(\mathcal{C}, \mathcal{V}, \mathcal{P}, \mathcal{F})$ of mutually disjoint sets of symbols, such that: $\mathcal{C}$ is a set of constant symbols, $\mathcal{V}$ is a set of variable symbols, $\mathcal{P}$ is a set of predicate symbols of finite arity, and $\mathcal{F}$ is a set of function symbols of finite arity.

For any $\mathcal{L}' \subseteq \mathcal{L}$, and any $\mathcal{V}' \subseteq \mathcal{V}$, we introduce the following notation:

- $\text{terms}(\mathcal{L}') \equiv \{ t \mid t \text{ is a term in } \mathcal{L}' \}$, e.g., if $\phi = P(f(a), b) \in \mathcal{L}$ we have $\{\phi\} \subseteq \mathcal{L}$ and $\text{terms}(\{\phi\}) = \{a, b, f(a)\}$.
- $\text{fterms}(\mathcal{L}') \equiv \{ t \mid t \in \text{terms}(\mathcal{L}') \text{ and } t \text{ is a function term} \}$, e.g., if $\mathcal{L}' = \{P(f(g(b))), P'(h(x), c)\}$ then $\text{fterms}(\mathcal{L}') = \{f(g(b)), g(b), h(x)\}$.

1 When $\mathcal{L}'$ is a singleton, e.g., $\mathcal{L}' = \{\phi\}$, we will write $\text{terms}(\phi)$ in place of $\text{terms}(\{\phi\})$, also: $\text{fterms}(\phi)$, $\text{atoms}(\phi)$, etc.
• atoms($L'$) $\equiv \{ a \mid a \text{ is any atomic predicate in } L' \}$, e.g., if $\phi = \alpha_1 \land \cdots \land \alpha_n$ is a conjunction of atoms: atoms($\phi$) $\equiv \{ \alpha_1, \ldots, \alpha_n \}$;
• literals($L'$) $\equiv \{ l \mid l \text{ is any literal in } L' \}$;
• $L'[V']$ $\equiv \{ \phi \mid \phi \in L' \land \phi \text{ contains no symbol in } V' \}$. $L[V']$ contains all the formulas of $L$ whose variable symbols are restricted to be necessarily in $V'$, e.g., given $\{ x, y \} \subseteq V$ if a formula $\phi \in L[\{ x, y \}]$; it may only contain $x$ and $y$ as free variables. Notice that $L[\emptyset]$ contains exactly the ground formulas of $L$.
• A substitution $\theta$ is any total function $\theta: V' \rightarrow \text{terms}(L[V \setminus V'])$ for any $V' \subseteq V$. For any formula $\phi \in L$, we write $\phi\theta$ for the formula that results after the substitution $\theta$ is applied to $\phi$, i.e., after replacing any free variable of $\phi$ that is also in the domain of $\theta$ for the corresponding term given by $\theta$. When $\theta: V' \rightarrow C$ we say that it is a substitution by constants.

2.2. PKS: the heir of STRIPS for incomplete information

PKS is similar to STRIPS in the sense that it models actions as the application of instantiated operators.

In STRIPS, an instantiated operator can be applied in a certain state of the world only when its preconditions hold at that state. When this happens it is said that the instantiated operator is executable. The application of an executable instantiated operator has the effect of changing the state of the world. PKS is similar to STRIPS, but it does not reason in terms of states of the world. It reasons instead in terms of the knowledge of the agent about the states of the world. In PKS, an instantiated operator can be applied only when its preconditions hold for the knowledge the agent has about the state of the world. When this happens it is said that the instantiated operator is executable. The application of an executable instantiated operator has the effect of changing the knowledge of the agent about the state of the world.

STRIPS is defined by means of a first-order language, i.e., by giving a grammar that defines how to write terms, literals, atoms and WFFs based on a tuple $(C, V, P, F)$. This first-order language is used to describe the states of the world, the domain’s action operators’ preconditions and effects, the initial state of the world, and a goal condition, (see [5,14,16,21]). PKS follows a similar pattern, it also uses a tuple of mutually disjoint sets of symbols $(C, V, P, F)$ to define a first order language, which in turn is used to describe the knowledge of the agent about the state of the world, the domain’s action operators’ preconditions and effects, the initial knowledge of the agent about the world, and a goal condition.

STRIPS usually represents the current state for which it is planning, as the contents of a single database with all the predicates about the world holding in that state. PKS uses three specialized databases: $K_f$, $K_w$, and $K_v$ to save a representation of the knowledge of the agent about the current state of the world for which it is planning. $K_f$ stores the facts about the world that will be known by the agent at that state, e.g., all the predicates that the agent knows will hold at that state. $K_w$ and $K_v$ store the knowledge about what the agent will have sensed before and at that state. $K_w$ is used to store all predicates about the world sensed at that state. The agent will know at that state whether the predicate holds or not because it has already sensed it. The formulas in $K_v$ are used to split the plan under
construction depending on the two possible outcomes of a sensing action. $K_v$ is like $K_w$, but it stores functions that the agent has already sensed at that state, functions that return constants, not formulas that can be true or false. The contents of the three databases can be consulted by the preconditions and modified by the effects of the instantiated operators.

The representation of the knowledge of the agent about the state of the world as the contents of three databases introduced above is one of the major contributions of PKS, as it allows us to reason directly in terms of these contents.

2.3. The database-state space and knowledge-state space

In this section we define a database-state, characterized by any possible content of the three databases $K_f$, $K_w$, and $K_v$, and its associated space. Afterwards, we show how the knowledge-states of the agent are derived from the database-states.

In the next definition we introduce three languages $\mathcal{L}_f$, $\mathcal{L}_w$, and $\mathcal{L}_v$, such that their formulas represent a valid entry for the corresponding database. We also define some auxiliary languages $\mathcal{L}_{l_0}$, $\mathcal{L}_{eq}$ and $\mathcal{L}_l$, used in Definition 1 and throughout the paper (e.g., Definitions 8 and 10). As usual, for any two terms $t, t'$ in terms($\mathcal{L}$), we write $t = t'$ instead of $= (t, t')$ for the equality binary predicate and $(t \neq t')$ as an alias for $\neg(t = t')$.

Definition 1. Given a first order language $\mathcal{L}$ with equality defined using the tuple $(\mathcal{C}, \mathcal{V}, \mathcal{P}, \mathcal{F})$ of mutually disjoint sets of constant, variable, predicate and function symbols, we define the following subsets of $\mathcal{L}$:

\[
\begin{align*}
\mathcal{L}_w &\equiv \{ \phi \mid \phi \text{ is any conjunction of atomic formulas of } \mathcal{L} \}, \\
\mathcal{L}_v &\equiv \{ f \mid f \in \text{ terms}(\mathcal{L}) \land \text{ terms}(f) \cap \text{ terms}(\mathcal{L}) = \emptyset \}, \\
\mathcal{L}_{l_0} &\equiv \{ \ell \mid \ell \in \text{ literals}(\mathcal{L}) \land \text{ terms}(\ell) \cap \text{ terms}(\mathcal{L}) = \emptyset \}, \\
\mathcal{L}_{eq} &\equiv \{ \ell \mid (\ell = (f = c) \lor \ell = (f \neq c)) \land f \in \mathcal{L}_v \land c \in \mathcal{C} \}, \\
\mathcal{L}_l &\equiv \mathcal{L}_{l_0} \cup \mathcal{L}_{eq}, \\
\mathcal{L}_f &\equiv \mathcal{L}_l[\emptyset].
\end{align*}
\]

$\mathcal{L}_v$ consists of function terms of $\mathcal{L}$ whose terms cannot be functions. The terms of any $f \in \mathcal{L}_v$ can be constants or variables but not functions, e.g., for $c \in \mathcal{C}$, $x \in \mathcal{V}$ and $f, g \in \mathcal{F}$ we have that $f(c, x) \in \mathcal{L}_v$ but $f(g(b)) \notin \mathcal{L}_v$. Similarly, $\mathcal{L}_{l_0}$ consists of literals of $\mathcal{L}$ whose terms cannot be functions. $\mathcal{L}_{eq}$ contains equality and inequality binary predicates whose left hand term is a function term in $\mathcal{L}_v$ and whose right hand term is a constant term. $\mathcal{L}_l$ consists of the union of $\mathcal{L}_{l_0}$ and $\mathcal{L}_{eq}$, and $\mathcal{L}_f$ consists of the ground formulas of $\mathcal{L}_l$, i.e., $\mathcal{L}_f[\emptyset]$. Notice that when $(f(t_1, \ldots, t_n) = t_{n+1}) \in \mathcal{K}_f$ all $t_i$ are variables or constants, not functions. In the original presentation of PKS [3,24] the equality expressions $(f = c) \in \mathcal{K}_f$ are called specifications of function values.

Strings in $\mathcal{L}_f$ are elements of $\mathcal{K}_f$; hence the power set of $\mathcal{L}_f$, denoted by $2^{\mathcal{L}_f}$, represents all the possible configurations of the database $\mathcal{K}_f$. Similarly $2^{\mathcal{L}_v}$ and $2^{\mathcal{L}_w}$ represent all the possible configurations of the other two databases $\mathcal{K}_w$ and $\mathcal{K}_v$. So, any state of the whole database can be represented simply by the cartesian product of the
three power sets that represent the contents of each database. The space of all possible database contents, called the database-state space and denoted by \( DS \), is defined as 
\[
DS \equiv 2^{L_f} \times 2^{L_w} \times 2^{L_v}.
\]
This space represents all possible configurations of the three databases \( K_f, K_w, \) and \( K_v \).

Therefore, any database-state \( d \in DS \) is a triple \( d = (d_f, d_w, d_v) \) that represents a particular content of the three databases. We denote by \( ||d|| \equiv |d_f| + |d_w| + |d_v| \) the total number of formulas present in state \( d \). As usual we denote by \( |S| \) the cardinality of any set \( S \), and therefore \( |d_f| \) is the number of formulas present in database \( K_f \). Similarly \( |d_w| \) and \( |d_v| \) with respect to \( K_w \) and \( K_v \).

We denote by \( L_{MLK} \) the language of the standard First-Order Modal Logic of Knowledge (MLK) introduced in [18] whose semantics is given in [20], basically an extension of the First-Order Logic with a modal Knowledge operator \( K \).

**Definition 2.** \( T_{MLK} : DS \to L_{MLK} \) is the Translation to MLK total function that maps any element \( d = (d_f, d_w, d_v) \in DS \) of the database-state space into some set of formulas in the First Order Modal Logic of Knowledge, through
\[
T_{MLK}(d) \equiv \left( \bigwedge_{\forall \ell(\vec{c}) \in d_f} K(\ell(\vec{c})) \right) \land \\
\left( \bigwedge_{\forall \phi(\vec{y}) \in d_w} \forall \vec{y}. K(\phi(\vec{y})) \lor K(\neg \phi(\vec{y})) \right) \land \\
\left( \bigwedge_{\forall f(\vec{y}) \in d_v} \forall \vec{y}. \exists v. K(f(\vec{y}) = v) \right).
\]

The function \( T_{MLK} \) allow us to interpret the contents of the databases of PKS as a statement in \( L_{MLK} \), which in turn has a formal semantics given in [20].

**Definition 3.** We define the language \( L_K \), called the PKS’s Knowledge Logic, as the range of the total function \( T_{MLK} \). Formally: \( L_K \equiv \text{range}(T_{MLK}) \).

Clearly, \( L_K \) is strictly contained in \( L_{MLK} \), i.e., \( L_K \subset L_{MLK} \).

**Definition 4.** The MLK-Entailment Equivalence Relation over the elements of the set of phrases in the PKS’s Knowledge Logic, denoted by \( \sim_{\models} \subseteq L_K \times L_K \), is defined for any pair \( w, w' \in L_K \) and where \( \models \) is the MLK entailment relation, by
\[
w \sim_{\models} w' \iff (w \models w' \land w' \models w).
\]

Notice that this relation is reflexive, symmetric and transitive, and therefore it is indeed an equivalence relation that induces a partition into equivalence classes over the strings in \( L_K \). We denote each equivalence class as usual by:
\[
[w]_{\sim_{\models}} \equiv \{w' \mid w \sim_{\models} w'\}.
\]

**Definition 5.** The set of all equivalence classes is called the Knowledge-State Space and is denoted by \( KS \), i.e.:
\[
KS = \{k \mid k = [w]_{\sim_{\models}} \land w \in L_K \}.
\]
Fig. 1. Two distinct database-states ending into a single knowledge-state.

Any element of the Knowledge-State Space set, i.e.: any $k \in KS$, is called a knowledge-state of the PKS’s framework.

**Remark 6.** Distinct database-states can correspond to the same knowledge-state, but two distinct knowledge-states always correspond to distinct database-states.

In fact, two distinct elements of the language $L_K$ do not necessarily correspond to two distinct knowledge-states, and can therefore collapse into a single knowledge-state because they may both belong to the same equivalence class. Two distinct database-states that are mapped by the function $T_{MLK}$ into two distinct strings in $L_K$ can belong to the same equivalence class and therefore be mapped into the same knowledge-state, see Fig. 1.

A simple example of this situation occurs in the following case: We have the following two distinct database-states $d = ((\alpha(c)), \{\alpha(c) \land \beta(y)\}, \emptyset) \in DS$ and $d' = ((\alpha(c)), \{\beta(y)\}, \emptyset) \in DS$ whose respective images under the translating function $T_{MLK}$ are $w = T_{MLK}(d) = K(\alpha(c)) \land \forall y.(K(\alpha(c) \land \beta(y)) \lor K(\neg(\alpha(c) \land \beta(y))))$ and $w' = T_{MLK}(d') = K(\alpha(c)) \land \forall y.(K(\beta(y)) \lor K(\neg\beta(y)))$. Both $w$ and $w'$ are capable of entailing the same set of formulas and therefore both belong to the same equivalence class, i.e., both represent the same knowledge-state $k = [w]_m = [w']_m \in KS$.

Obviously, any two distinct knowledge-states $[w]_m \neq [w']_m$ always correspond to two distinct database-states $w \neq w'$, because otherwise, by supposing $w = w'$, we get $[w]_m = [w']_m$, which is absurd.

**2.4. The PKS planning problem**

The purpose of this section is to introduce the definition of an instance of the PKS planning problem. This definition relies on three languages that we have not yet introduced: the query language (used to specify the goal), the operator specification language and the domain specification language. Hence, we start by defining these languages.
Definition 7. Let the parameterized query language $L(G_{pq})$ be the language produced by the grammar $G_{pq}$ shown in Table 1. We define the query language $L_q$ as the subset of $L(G_{pq})$ that contains only ground formulas; formally

$$L_q \equiv L(G_{pq})[\emptyset].$$

The parameterized query language $L(G_{pq})$ allow us to specify parameterized families of conjunctions of possible negated primitive queries. The query language $L_q$ contains conjunctions of possible negated ground primitive queries. Notice that in the case of the $K_f(\ell)$ primitive queries the $\ell$ is in literals($L$), far more expressive than $L_f$ that determines the contents of $K_f$. literals($L$) includes variables, nested terms and also the case of equality $t = t'$ and inequality $t \neq t'$ predicates of any pair of terms $t, t' \in \text{terms}(L)$. Similarly, terms($L$) is more expressive than $L_v$.

Now we simultaneously introduce the definitions of $L_D$ and $L_O$, respectively, the domain specification language of PKS and the language for specifying its operators. As in STRIPS we introduce a new set $N$ of operator’s name symbols, mutually disjoint with the previously introduced ones ($C, V, P, F$).

Definition 8. The Domain and Operator Specification Languages are respectively defined by

$$L_D \equiv 2^{L_O} \setminus \emptyset, \quad \text{any non-empty set of operators specifications } \alpha \in L_O,$$

$$L_O \equiv \{ \alpha \mid \alpha = (n(\vec{x}), P(\vec{x}), (C_1(\vec{x}) \Rightarrow E_1(\vec{x}, \vec{y}), \ldots, C_r(\vec{x}) \Rightarrow E_r(\vec{x}, \vec{y}))) \},$$

where

- $n \in N$ is the operator’s name;
- $\vec{x}$ are the operator’s parameters: a possibly empty finite list of variable symbols. $V_{\vec{x}}(\alpha) \subseteq V$ denotes the finite set of variables that appear in $\vec{x}$;
- $\vec{y}$ are the free variables of the operator’s effect: a possibly empty finite list of variable symbols. $V_{\vec{y}}(\alpha) \subseteq V$ denotes the finite set of variables that appear in $\vec{y}$ and is such that $V_{\vec{x}}(\alpha) \cap V_{\vec{y}}(\alpha) = \emptyset$. Also: $V_{\vec{x} \vec{y}}(\alpha) \equiv V_{\vec{x}}(\alpha) \cup V_{\vec{y}}(\alpha)$;
- $P(\vec{x}) \in L(G_{pq})[V_{\vec{x}}(\alpha)]$ is called the operator’s precondition;
- The non-empty list of $C_i(\vec{x}) \Rightarrow E_i(\vec{x})$ is called the operator’s conditional-effect clause and is such that $C_i(\vec{x}) \in L(G_{pq})[V_{\vec{x}}(\alpha)]$ and $E_i(\vec{x}, \vec{y}) \in L(G_{pe})[V_{\vec{x} \vec{y}}(\alpha)]$ for all $i = 1, \ldots, r$ with $r > 0$;
- The rules for the grammars $G_{pq}$ and $G_{pe}$ are respectively given in Tables 1 and 2.

For any operator $\alpha = (n(\vec{x}), P(\vec{x}), (C_1(\vec{x}) \Rightarrow E_1(\vec{x}, \vec{y}), \ldots, C_r(\vec{x}) \Rightarrow E_r(\vec{x}, \vec{y})))$ in $L_O$ we introduce the following two notations: $\text{Pre}(\alpha)$ denotes the precondition of $\alpha$, and $\text{Eff}(\alpha)$ denotes the set of all its conditional-effects of $\alpha$, formally:

- $\text{Pre}(\alpha) \equiv P(\vec{x})$,$$
- \text{Eff}(\alpha) \equiv \{ C_1(\vec{x}) \Rightarrow E_1(\vec{x}, \vec{y}), \ldots, C_r(\vec{x}) \Rightarrow E_r(\vec{x}, \vec{y}) \}.$$


Table 1
Set of rules for the grammar $G_{pq}$ of the parameterized query language

<table>
<thead>
<tr>
<th>Rule</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$\rightarrow \neg Q</td>
</tr>
<tr>
<td>$Q$</td>
<td>$\rightarrow K_f(\ell)</td>
</tr>
<tr>
<td>$\ell$</td>
<td>$\rightarrow \ell \in \text{literals}(L)$</td>
</tr>
<tr>
<td>$\phi$</td>
<td>$\rightarrow \phi \in \mathcal{L}_w$</td>
</tr>
<tr>
<td>$f$</td>
<td>$\rightarrow f \in \text{terms}(L)$</td>
</tr>
</tbody>
</table>

Table 2
Set of rules for the grammar $G_{pe}$ of the parameterized effect language (sketch)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_f(\vec{x}, \vec{y})$</td>
<td>$\rightarrow E_f(\vec{x}, \vec{y})</td>
</tr>
<tr>
<td>$E_f(\vec{x}, \vec{y})$</td>
<td>$\rightarrow \text{add}(K_f, \ell(\vec{x}))</td>
</tr>
<tr>
<td>$E_f(\vec{x}, \vec{y})$</td>
<td>$\rightarrow \text{add}(K_w, \phi(\vec{x}, \vec{y}))</td>
</tr>
<tr>
<td>$E_f(\vec{x}, \vec{y})$</td>
<td>$\rightarrow \text{add}(K_v, f(\vec{x}, \vec{y}))</td>
</tr>
</tbody>
</table>

It is easy to observe that the syntax to specify a domain in PKS is more complex than in STRIPS. In the definition of $\mathcal{L}_O$ we use all the languages presented in Definition 1 and those produced by the grammars shown in Tables 1 and 2. This greater complexity is a consequence of: first, the presence of two disjoint sets of variables, the parameters $\vec{x}$ of the operator and the effect’s free variables $\vec{y}$, and not just one; second, the presence of a particular and different language for the preconditions and the conditions of the conditional-effects: $\mathcal{L}(G_{pq})$; third, the effects which are now conditional; and fourth, the presence of three distinct databases, that forces us to branch the syntax of the effects, depending on which database is affected by the effects, and not only depending on which $\text{add}$ or $\text{delete}$ operation is performed.

We can now finally introduce the definition of an instance of the PKS planning problem. This definition relies on the languages $\mathcal{L}_q$ and $\mathcal{L}_D$ previously introduced and on the database-state space $\mathcal{D}_S$ defined in the previous section.

**Definition 9.** The set of all instances of the PKS planning problem, denoted by $I_{PKS}$, is defined by $I_{PKS} \equiv \mathcal{L}_D \times \mathcal{D}_S \times \mathcal{L}_q$.

For any instance $x = (\mathcal{D}, d_0, \mathcal{G}) \in I_{PKS}$ we denote by:
1. $\mathcal{D}(x) \equiv \mathcal{D} \in \mathcal{L}_D$ its planning domain specification;
2. $d_0(x) \equiv d_0 \in \mathcal{D}_S$ its initial condition, given as an initial database-state;
3. $\mathcal{G}(x) \equiv \mathcal{G} \in \mathcal{L}_q$ its goal, given as a query.

Clearly, $x \in I_{PKS}$ means that $x$ is an instance of the PKS planning problem. We say that an instance $x = (\mathcal{D}, d_0, \mathcal{G}) \in I_{PKS}$ is finite iff all the sets $\mathcal{D}, d_{0f}, d_{0w}, d_{0v}$ and $\mathcal{G}$ are finite. This implies the number of operators used to specify the domain $|\mathcal{D}|$ and the number of formulas present at the initial state $||d_0||$ are finite.
It is worth noticing that the definition of an instance of the planning problem in the original presentation of PKS [24] includes a fourth component: $\mathcal{U}$, the set of domain specific knowledge update rules. We do not include it here because it is an implementation artifact not essential for the theoretical analysis of the framework, as we show in Section 2.6.

2.5. Solutions to the PKS planning problem

We now give the formal characterization of the solutions of an instance of the PKS planning problem. We start by giving a meaning to the queries expressed in the query language $\mathcal{L}_q$ that is used to write the preconditions, the conditions in the conditional-effects of the operators, and the goal condition. After that, we define when an operator is executable, and the results of applying it. Finally, we define a conditional plan and when one is considered a solution of an instance of the PKS planning problem.

As Table 1 shows, the query language $\mathcal{L}_q$ is just a conjunctive list of possible negated primitive queries $K_f(\ell), K_w(\phi)$ and $K_v(t)$, therefore, the semantics of $\mathcal{L}_q$ is determined by the semantics of those primitive queries. In [3], the meaning of these three types of primitive queries is given in terms of the output of an inference algorithm, called IA, which consults the contents of the three databases to determine their value. In contrast, in the next definition, we give their meaning by defining the recursive function $\eta$.

**Definition 10.** The total function $\eta: \mathcal{L}_q \times DS \rightarrow \{\top, \bot\}$, called the result of query $q$ at database-state $d$, is defined recursively for any state $d = (df, dw, dv) \in DS$ and $q, q', K_f(\ell), K_w(\phi), K_v(t) \in \mathcal{L}_q$ by

$$
\eta(K_f(\ell), d) = \begin{cases} 
\top, & \text{if } (\xi(\ell) = \alpha \land \alpha \in df) \lor (\xi(\ell) = \neg\alpha \land \neg\alpha \in df) \lor \\
\bot, & \text{o.w.,}
\end{cases}
$$

$$
\eta(K_w(\phi), d) = \begin{cases} 
\top, & \text{if } (\xi(\phi) \in df) \lor (\exists f \in dw \land \exists \theta : V \rightarrow C \cup \text{terms}(\xi(\phi))) \land \exists \alpha_j \in \text{atoms}(\psi) \\
\bot, & \text{if } (\forall \alpha_j \in (\text{atoms}(\psi) \setminus \alpha_i)).\eta(K_f(\alpha_j \theta), d) = \top \land \\
\bot, & \text{o.w.,}
\end{cases}
$$

$$
\eta(K_v(t), d) = \begin{cases} 
\top, & \text{if } (\xi(t) \in C) \lor (\exists f \in dv, \exists \theta : V \rightarrow C, f \theta = \xi(t)) \\
\bot, & \text{o.w.,}
\end{cases}
$$

$$
\eta(q \land q', d) = \eta(q, d) \land \eta(q', d),
$$

$$
\eta(\neg q, d) = \neg \eta(q, d),
$$

where $\xi(\ell)$ is defined recursively for any $\ell \in \text{literals}(\mathcal{L})[\emptyset]$ by

$$
\xi(\ell) = \begin{cases} 
\xi(\ell[\rightarrow c]), & \text{if } \exists f \in \text{terms}(\ell) \land \ell \in \mathcal{L}_f \land \exists f = c \in df, \\
\ell, & \text{o.w.}
\end{cases}
$$

The first three entries give the semantics of the three types of primitive queries, the value of $\eta(q, d)$ for the case in which $q$ is a primitive query: $q = K_f(\ell), K_w(\phi)$ or $K_v(t)$. The other two entries give the usual semantics for conjunction and negation of queries.
Generally speaking, the primitive queries answer \( \top \) when the queried formula has a counterpart in the corresponding database, and \( \bot \) otherwise. More specifically, in the case of \( K_f(\ell) \), any function term \( f \) present in the queried formula \( \ell \) is replaced by a constant \( c \) whenever an equality expression \( (f = c) \) is present in database \( K_f \). Function \( \xi \) does this recursive replacement in \( \ell \) of function terms by constants until there are no more function terms to replace. In the definition of \( \xi(\ell) \) the expression \( \ell[f \rightarrow c] \) denotes the substitution of every occurrence of term \( f \) in \( \ell \) by term \( c \). For example, being \( (f(a,b) = c) \) and \( (g(d) = b) \) both in \( df \), the value given by \( \xi(P(f(a,g(d))) \) is \( P(c) \) because in a first step \( g(d) \) is replaced by \( b \) and in a second step \( f(a,b) \) is replaced by \( c \). Hence, \( \xi \) tries to transforms any \( \ell \) in literals \( (L_f)[\emptyset] \), a language where nested function terms are allowed, into a literal without nested function terms, and so in \( L_f \), the language for the contents of \( K_f \) database. Afterwards, the transformed \( \xi(\ell) \) is checked against the contents of database \( K_f \). The trivial case of equality tautology is treated in the second line without any need of consulting \( K_f \). In the case of \( K_w(\phi) \) and \( K_v(t) \), the formulas in the corresponding databases can have variables, and therefore, the match is searched against any substitution of those variables (e.g., \( f\theta = \xi(t) \) in the \( K_v(\phi) \)). The case of \( K_w(\phi) \) is more complex than \( K_v(t) \) because the formulas \( \phi \) in \( K_w \) are conjunctions of atomic predicates, while those of \( K_v \) are single function terms.

The results of the analysis of PKS in Section 3 depend on the fact that the function \( \eta \) is completely determined by the current database-state, i.e., it has the form \( \eta: L_q \times DS \rightarrow \{\top, \bot\} \). We use the fact that \( \eta \) has this form to define when an instantiated operator is consistent and executable in a certain database-state. Notice that, even if we can give a slightly different semantics to the primitive queries by changing the definition of \( \eta \) without changing its form, the results of the analysis of PKS are independent of this variation.

**Definition 11.** Given an operator \( \alpha \in L_O \) and a substitution by constants \( \theta : V(\alpha) \rightarrow C \), we say that \( \alpha \theta \) is the instantiated operator that results from applying to \( \alpha \) substitution \( \theta \) which replaces all the parameters in \( \alpha \) by the corresponding constants given by \( \theta \).

**Definition 12.** For any instance \( x = (D, d_0, G) \in I_{PKS} \) and for any operator \( \alpha \in D \) we define the set \( E(\alpha) \), called the expansion of operator \( \alpha \), that contains all the possible instantiated operators derived from \( \alpha \) and the set \( ED(x) \), called the expansion of domain \( D \), with all the possible instantiated operators in \( x \). Formally:

\[
E(\alpha) \equiv \{ \alpha \theta \mid \theta : V(\alpha) \rightarrow C \},
\]

\[
ED(x) \equiv \bigcup_{\forall \alpha \in D} E(\alpha).
\]
allowed to be bound to any ground function term, any operator \( A(x) \in L_\Omega \) that has an effect \( \text{add}(K_f, \phi(x)) \) can add a formula to \( K_f \) that violates the type of contents that \( K_f \) can hold: It will add \( \phi(f(a)) \) when instantiated with the ground function \( f(a) \). Definition 1 states that \( K_f \) cannot have formulas whose terms are other than constants.\(^2\)

We now state the conditions for operators to be consistent and executable. Before doing this, however, it is necessary to introduce the following six sets, whose elements are the formulas respectively added and deleted to each one of the three databases \( K_f, K_w \) and \( K_v \) by the conditional-effects of any instantiated operator \( \alpha\theta \in ED \) in any database-state \( d \in DS \):

- \( \text{Eff}_f^+ (\alpha\theta, d) \equiv \{ \ell \mid \ell \in L_f \land \exists (C \Rightarrow E) \in \text{Eff}(\alpha\theta) \land E = \text{add}(K_f, \ell) \land \eta(C, d) = \top \} \),
- \( \text{Eff}_f^- (\alpha\theta, d) \equiv \{ \ell \mid \ell \in L_f \land \exists (C \Rightarrow E) \in \text{Eff}(\alpha\theta) \land E = \text{delete}(K_f, \ell) \land \eta(C, d) = \top \} \),
- \( \text{Eff}_w^+ (\alpha\theta, d) \equiv \{ \phi \mid \phi \in L_w \land \exists (C \Rightarrow E) \in \text{Eff}(\alpha\theta) \land E = \text{add}(K_w, \phi) \land \eta(C, d) = \top \} \),
- \( \text{Eff}_w^- (\alpha\theta, d) \equiv \{ \phi \mid \phi \in L_w \land \exists (C \Rightarrow E) \in \text{Eff}(\alpha\theta) \land E = \text{delete}(K_w, \phi) \land \eta(C, d) = \top \} \),
- \( \text{Eff}_v^+ (\alpha\theta, d) \equiv \{ f \mid f \in L_v \land \exists (C \Rightarrow E) \in \text{Eff}(\alpha\theta) \land E = \text{add}(K_v, f) \land \eta(C, d) = \top \} \),
- \( \text{Eff}_v^- (\alpha\theta, d) \equiv \{ f \mid f \in L_v \land \exists (C \Rightarrow E) \in \text{Eff}(\alpha\theta) \land E = \text{delete}(K_v, f) \land \eta(C, d) = \top \} \).

**Definition 13.** Given any instance \( x = (D, d_0, G) \in IPKS \) of the PKS planning problem, for any instantiated operator \( \alpha\theta \in ED(x) \) and for any database-state \( d \in DS \), we say that:

- \( \alpha\theta \) is **consistent** in database-state \( d \) iff:\(^3\)
  \[
  \text{Eff}_f^+ (\alpha\theta, d) \cap \text{Eff}_f^- (\alpha\theta, d) = \emptyset,
  \text{Eff}_w^+ (\alpha\theta, d) \cap \text{Eff}_w^- (\alpha\theta, d) = \emptyset,
  \text{Eff}_v^+ (\alpha\theta, d) \cap \text{Eff}_v^- (\alpha\theta, d) = \emptyset,
  +\text{Eff}_f (\alpha\theta, d) \cap \eta(Pre(\alpha\theta), d) = \emptyset;
  \]
- \( \alpha\theta \) is **executable** in database-state \( d \) iff:
  \[
  \alpha\theta \text{ is consistent in } d \land \eta(Pre(\alpha\theta), d) = \top.
  \]

When an instantiated operator \( \alpha\theta \) is executable in a database-state \( d \in DS \) we also say that \( \alpha \) is \( \theta \)-**executable** in that database-state \( d \).

\(^2\) In [3] (see the “Open Safe Domain” example) the authors seem to allow the use of more general terms at least in a limited form.

\(^3\) For any \( L \subseteq \text{lit}(L) \) we denote \( +L \) and \( -L \) respectively the sets of positive and negative literals of \( L \), i.e.,
  \[ +L = \{ \ell \mid \ell \in L \land \ell = a \land a \in \text{atoms}(L) \} \]
  \[ -L = \{ \ell \mid \ell \in L \land \ell = -a \land a \in \text{atoms}(L) \} \].
Definition 14. Given a substitution $\theta$ and an operator $\alpha \in \mathcal{L}_O$ such that $\alpha$ is $\theta$-executable in database-state $d \in \mathcal{DS}$; we say that applying the executable instantiated operator $\alpha\theta$ in database-state $d$ results in the database-state $d' = (d'_f, d'_u, d'_v) \in \mathcal{DS}$, denoted as $d \models^{\alpha\theta} d'$, iff:

$$d'_f = d_f \cup \text{Eff}^+_f(\alpha\theta, d) \setminus (\text{Eff}^-_f(\alpha\theta, d) \cup \text{Eff}^+_f(\alpha\theta, d)),$$

$$d'_u = d_u \cup \text{Eff}^+_u(\alpha\theta, d) \setminus \text{Eff}^-_u(\alpha\theta, d),$$

$$d'_v = d_v \cup \text{Eff}^+_v(\alpha\theta, d),$$

where $\text{Eff}^+_f(\alpha\theta, d) \equiv \{a | \alpha \in \text{Eff}^+ f(\alpha\theta, d)\} \cup \{\neg a | a \in \text{Eff}^+ f(\alpha\theta, d) \cap \text{atoms}(\mathcal{L})\}$. We refer to $d'$ by $\text{Result}(\alpha\theta, d)$.

Notice that the result $d'_f$ is defined in a slightly different form. The set of literals $\text{Eff}^+_f(\alpha\theta, d)$ is deleted from $d_f$ to avoid an inconsistent $d_f$ with an atom and its negation. This is done by deleting from $d_f$ the negation of every literal that it is added to.

In any database-state $d = (d_f, d_u, d_v) \in \mathcal{DS}$ such that $d_u \neq \emptyset$ we can use any formula $\phi \in d_u$, to split the database-state $d$ into the two possible outcomes of the sensing of $\phi$, i.e., the database-states $d'$ and $d''$ defined by $d' = (d_f \cup \{\phi\}, d_u \setminus \{\phi\}, d_v)$ and $d'' = (d_f \cup \{\neg \phi\}, d_u \setminus \{\phi\}, d_v)$. In this case we say that $d'$ is the positive outcome of sensing $\phi$ at state $d$, denoted as $d \models^\phi d'$, and correspondingly $d \models^{\neg \phi} d''$ denotes the negative outcome of sensing $\phi$ at the same state $d$. We refer to $d'$ by $\text{Result}^+(\phi, d)$, and to $d''$ by $\text{Result}^-(\phi, d)$. Recall that $\phi(\tilde{y}) \in d_u$ means we have sensed $\phi(\tilde{y})$ in database-state $d$ and therefore the agent knows whether $\phi(\tilde{y})$ or its negation holds for any value of its free variables: $\forall \tilde{y}.K(\phi(\tilde{y})) \lor K(\neg \phi(\tilde{y}))$.

Given a directed graph $G = (V, E)$ where $V$ is the set of nodes and $E \subseteq V \times V$ is the set of edges, we denote by $V(G)$ its set $V$ and by $E(G)$ its set $E$. Also we denote: $T(G) \subseteq V$ the set of terminal nodes of $G$ defined by $T(G) \equiv \{n | n \in V \wedge \forall n' \in V.(n, n') \notin E\}$.

Definition 15. Given an instance $x = (D, d_0, G) \in I_{PKS}$ of the PKS planning problem, a conditional plan $P$ is defined as any directed graph $G$ such that the nodes are a subset of the database-states $V(P) \subseteq \mathcal{DS}$, $d_0 \in V(P)$ and any edge $d, d' \in E$ comply with any of the following three conditions:

1. $d \models^{\alpha\theta} d'$ for any $\alpha \in \mathcal{L}_O$ that is $\theta$-executable in state $d$;
2. $d \models^\phi d'$ for any $\phi \in d_u$, and there also exists another edge $e' \in E$ that connects $d$ with a node $d'' \in V(P)$, and $d \models^{\neg \phi} d''$;
3. $d \models^{\neg \phi} d'$ for any $\phi \in d_u$, and there also exists another edge $e' \in E$ that connects $d$ with a node $d'' \in V(P)$, and $d \models^{\phi} d''$.

We denote as plans($d_0, D$) the set of conditional plans in $D$ starting in $d_0$.

Notice that the directed graph that represents a conditional plan is such that it branches when splitting a database-state into the two positive and negative outcomes of a sensed formula at that state, and grows by means of applying an executable instantiated operator.
αθ at the corresponding state represented by the node. We denote by $\models^*_P$ the reflexive transitive closure of the binary relation between database-states defined by $P$, thus, $d \models^*_P d'$ means that there exists a path in $P$ from $d$ to $d'$ or that $d = d'$.

Fig. 2(a) shows an example of a conditional acyclic plan $P$ for an instance $x = (D, d_0, G) \in IPKS$. Notice that the edges are completely characterized by the $\models \cdots$ symbols denoting which operation produces the corresponding change of state.

It is possible to define different kinds of valid solutions of an instance of the PKS planning problem depending on properties we require from the conditional plan $P$. We can accept $P$ as a valid plan whether it is a cyclic directed graph or not. We can require that the goal is satisfied in any terminal node of $P$ or just in one of them. Therefore, we can define four kinds of solutions:

Definition 16. Given an instance $x = (D, d_0, G) \in IPKS$ of the PKS planning problem we say that a conditional plan $P$ is a:

1. **Strong [Acyclic] Solution of $x$** iff $P$ is acyclic $\land \left( \forall d \in T(P). \eta(G, d) = T \right)$.

2. **Strong Cyclic Solution of $x$** iff $\left( \forall d \in V(P). \exists d' \in T(P). d \models^*_P d' \right) \land \left( \forall d \in T(P). \eta(G, d) = T \right)$.

3. **Weak Acyclic Solution of $x$** iff $P$ is acyclic $\land \left( \exists d \in T(P). \eta(G, d) = T \right)$.

4. **Weak Cyclic Solution of $x$** iff $\exists d \in T(P). \eta(G, d) = T$.

We denote by $SOL^+_PKS(x)$, $SOL^-_{PKS}(x)$, $SOL^+_{PKS}(x)$ and $SOL^-_{PKS}(x)$ the sets of all conditional plans $P$ that are respectively strong, strong cyclic, weak acyclic and weak cyclic solutions of $x$. Also, we use $SOL_{PKS}(x)$ meaning any of the previous four cases, $SOL^+_{PKS}(x) \equiv SOL^+_{PKS}(x) \cup SOL^-_{PKS}(x)$ and $SOL^-_{PKS}(x) \equiv SOL^-_{PKS}(x) \cup SOL^-_{PKS}(x)$.

$P$ is a strong acyclic solution, or simply a strong solution, if it has no cycle and $G$ holds in all its terminal nodes. $P$ is a weak acyclic solution if it is also not cyclic and $G$ holds in at least one terminal node. $P$ is a strong cyclic solution if it may be cyclic but $G$ holds in all its terminal nodes, and from every database-state $d$ in $V(P)$ there is a path to a terminal node $d'$, i.e., if the condition $\left( \forall d \in V(P). \exists d' \in T(P). d \models^*_P d' \right)$ holds. This condition implies that we accept a cyclic graph as a strong cyclic solution only if it is always possible for the execution to terminate; this condition excludes “dead-end” cycles, i.e., cycles composed of nodes from which there is no path to a terminal of $P$. Therefore, if execution leads to such a cycle it is doomed to loop endlessly. $P$ is a weak cyclic solution if it is cyclic and has at least one terminal node in which $G$ holds. Notice that weak cyclic solutions include the case of $P$ having “dead-end” cycles.

Fig. 2 shows four conditional plans that are respectively (a) strong, (b) strong cyclic, (c) weak acyclic and (d) weak cyclic solutions. Terminals that satisfy the goal condition are depicted with double circles, e.g., the double circles in (a) mean that $\eta(G, d_6) = \eta(G, d_7) = \eta(G, d_9) = T$. For clarity we omit most of the labeling from transitions and database-states when not necessary. The solution in (c) is weak because the goal condition holds only in
one of its terminals. The weak cyclic solution in (d) clearly shows both causes that separate this class from the strong cyclic solutions: first, it has terminals that do not satisfy the goal condition, and second, it also has a “dead-end” cycle (the shorter cycle) from which it is impossible to reach the terminals of the plan. The numbers in (d) are used later in the paper.

The original version of PKS presented in [24] defines a conditional plan as a tree (no cycles) and a solution as any plan such that the goal is satisfied in all its leaves. Thus, it belongs to the class of strong acyclic solutions.

Many non-probabilistic frameworks for planning with incomplete information and sensing deal only with strong solutions. They search for strong solutions and when it is not possible to find one, they simply fail to provide any solution. This is the case with the original PKS framework [24]. It is worth highlighting that our definition of the semantics of PKS has easily allowed us to extend the original proposal of PKS to more general classes of solutions. The definition of conditional plans as directed graphs instead of trees allows us to define four kinds of solutions: strong, strong cyclic, weak acyclic, weak cyclic (Definition 16). Furthermore, this extension is important for any subsequent comparison of PKS with other frameworks for planning with incomplete information and sensing because most of them represent conditional plans as directed graphs. The distinction between plans that are strong, strong cyclic and weak solutions of the planning problem was first introduced in [8] in the context of Planning as Symbolic Model Checking. It was formalized later in [10]. For a formal account with proofs and experimental evaluation see [7]. It is important to notice that in those articles there is no distinction between weak acyclic and weak cyclic solutions; both are subsumed by a unique weak solution class. As we shall see in Section 4, this distinction is important.

2.6. Some remarks on the proposed formalization of PKS

We explain here why we left outside of the present formalization some characteristics of the original presentation of PKS.
PKS has two mechanisms designed to facilitate the specification of action operators that are more complex than in STRIPS, because, when specifying the add/delete effects of the action, it is needed to keep consistent the contents of three databases instead of just one. These mechanisms are the update rules, that are specific for each domain and the consistency rules, that are domain independent.

In [24] it is introduced a new database, $K_x$, not present in [3]. $K_x$ is used to represent a fourth type of knowledge, called ‘exclusive-or’ knowledge, that cannot be represented with the contents of the other three databases. It is worth remarking that in [24] the IA algorithm it is not updated to use $K_x$ knowledge and that from the point of view of planning $K_x$ is used only to implement the consistency rules. As explicitly stated in those papers, the consistency rules are not necessary, but just a convenient mechanism to simplify the specification of actions. It is always possible to describe actions’ effects so as to replace these rules entirely. Hence, for the sake of simplicity in the present paper we have not modeled the update rules, some of the consistency rules, and $K_x$, considering they are not an indispensable part of PKS for planning. 4

As it is described in [3], PKS can also be used for execution, not just for planning, and for that purpose it includes an action’s run-time effect part in its specification and also a specialized fifth database: $LCW$, the Local Closed Word database, that is the analog of $K_w$ for run-time. As we said, here we are interested in analyzing only this framework for planning purposes, hence, in what follows, we ignore the $LCW$ database. 5

Hence, dropping the update rules, some of the consistency rules, $K_x$ and $LCW$, does not make the current formal version of PKS less powerful from the point of view of planning. We can parallel in the present version of PKS every planning behaviour of the original PKS. Finally, we cannot finish this section without emphasizing that in practice both $K_x$ and $LCW$ are extremely important to simplify the modeling of domains. They allow us to express, in an extremely synthetic form, knowledge that cannot always be represented statically with the other databases, but only by tinkering with the actions’ specification.

3. An analysis of the PKS framework

In Section 2 we introduced a formal semantics for PKS. Here we use it to show some interesting properties of PKS and afterwards, we use these results to introduce a complete and sound planning algorithm for PKS.

3.1. The PKS database-state space

Given any instance $x \in I_{PKS}$ of the PKS planning problem, we define the set $DS(x) \subseteq DS$, called the reachable database-state space for $x$, by

$$DS(x) \equiv \{ d \in DS \land \exists P \in \text{plans}(d_0(x), D(x)), d_0(x) \models P d \}.$$

4 We model some of the consistency rules in Definition 13. Also Definition 14 guarantee the first of the these rules.

5 In this we follow [24] which also deals with planning exclusively.
This subset of $\mathcal{DS}$ contains all the possible database-states that can be reached starting from the initial state of $x$ using the reasoning mechanism of PKS for the domain $D$, i.e., by applying any combination of $\theta$-executable operators $\alpha \in D$ or splitting of database-states by using formulas in $K_w$. $\mathcal{DS}(x)$ is the space where the search for a solution for the instance $x$ is done.

**Theorem 17.** Given a finite instance $x = (D, d_0, \mathcal{G}) \in I_{PKS}$ of the PKS planning problem their corresponding reachable database-state space $\mathcal{DS}(x)$ is finite and bounded by

$$v |\mathcal{DS}(x)| \leq 2^{|D| |C| \max_{\alpha \in D} |V_\alpha(\alpha)|} \in O(|C|!),$$

with

- $|D|$ the number of operators in the domain $D$;
- $|C|$ the number of constants;
- $m_{\Phi}$ is the maximum number of add effects over a single database by a single operator in $D$ (formal definition below in Eq. (4));
- $\|d_0\| = |d_{0_f}| + |d_{0_w}| + |d_{0_v}|$, the total number of formulas present in $d_0$;
- $m_{\bar{x}} \equiv \max_{\alpha \in D} (|V_\alpha(\alpha)|)$, the maximum arity of an operator of $D$.

**Proof.** To prove this theorem we need to bound the number of the database-states in the reachable database-state space for the instance $x \in I_{PKS}$. Because a database-state is by definition determined by the contents of the three databases we proceed by bounding the number of the formulas that can be added to the databases by means of the application of any operator $\alpha \in D$. By hypothesis the number of formulas initially present in the databases is finite and equal to $\|d_0\|$. The operators can possibly have parameters and, as it was explained in Section 2.5, any formula that can be added to the databases when applying the results of an instantiated operator has those parameters substituted by constants. Therefore the set $E_D(x)$ with all the possible instantiated operators in the domain (see Definition 12) contains all the formulas that can be added to the databases. If we add to these formulas the formulas initially present in the database we have an upper bound to all the possible formulas that can be present in any of the three databases. Therefore, the cartesian product of the power set of each of these three sets of formulas certainly includes any possible database-state that can be reached from $d_0$ in the domain $D$ of instance $x$. The preconditions, the conditional part of the effects and the delete effects are the mechanism by which PKS generates only some of these database-states, those that respect the rules of the particular domain $D$, and not every possible combinations that the power set represents.

Therefore, the proof proceeds by first bounding the number of instantiated operators in $E_D(x)$. For any $\alpha \in D$ we have that

$$|E(\alpha)| = \left[ \frac{|C|}{|V_\alpha(\alpha)|} \right] \leq \left[ \frac{|C|}{m_{\bar{x}}} \right] \in O(|C|!),$$

where $\left[ \frac{n}{k} \right]$ is the Stirling number that counts the number of arrangements of $k$ elements taken from a set of $n$ elements, and with $m_{\bar{x}}$ the maximum arity of the operators in $D$ defined in Theorem 17.
Therefore, using (1), a bound for the number of elements in $E^D(x)$ is

$$|E^D(x)| = \bigg| \bigcup_{\alpha \in D} E(\alpha) \bigg| \leq \sum_{\alpha \in D} |E(\alpha)| \leq |D| \left[ \frac{|C|}{m_x^D} \right] \in \mathcal{O}(|D||C|).$$  (2)

For any instantiated operator $\alpha \theta \in E^D(x)$ we define three different sets $\Phi_f(\alpha \theta), \Phi_w(\alpha \theta)$ and $\Phi_v(\alpha \theta)$ such that each collects all the formulas that can be potentially added to the corresponding database. Formally:

$$\Phi_f(\alpha \theta) \equiv \{ \ell \mid \exists (C \Rightarrow E) \in \text{Eff}(\alpha \theta) \wedge E = \text{add}(K_f, \ell) \},$$

$$\Phi_w(\alpha \theta) \equiv \{ \phi(\vec{y}) \mid \exists (C \Rightarrow E) \in \text{Eff}(\alpha \theta) \wedge E = \text{add}(K_w, \phi(\vec{y})) \},$$

$$\Phi_v(\alpha \theta) \equiv \{ f(\vec{y}) \mid \exists (C \Rightarrow E) \in \text{Eff}(\alpha \theta) \wedge E = \text{add}(K_v, f(\vec{y})) \}.$$

We now define the three sets that contain all the formulas that can be added by any of the instantiated operators to each of the three databases:

$$\Phi_f \equiv \bigcup_{\alpha \theta \in E^D(x)} \Phi_f(\alpha \theta), \quad \Phi_w \equiv \bigcup_{\alpha \theta \in E^D(x)} \Phi_w(\alpha \theta),$$

$$\Phi_v \equiv \bigcup_{\alpha \theta \in E^D(x)} \Phi_v(\alpha \theta).$$

With these sets it is easy to define the three sets that contain all the formulas that can ever be present in the different databases that compose $DS$ for the instance $x$, simply the union of each of these sets with the corresponding set of formulas initially in the database-state $d_0 = (d_{0_f}, d_{0_w}, d_{0_v})$:

$$\Phi^*_f \equiv \Phi_f \cup d_{0_f}, \quad \Phi^*_w \equiv \Phi_w \cup d_{0_w}, \quad \Phi^*_v \equiv \Phi_v \cup d_{0_v}.$$

Using (2) we have that for $\xi = f, w, v$

$$|\Phi_\xi| \leq \bigg| \bigcup_{\alpha \theta \in E^D(x)} \Phi_\xi(\alpha \theta) \bigg| \leq \sum_{\alpha \theta \in E^D(x)} |\Phi_\xi(\alpha \theta)| \leq |D| \left[ \frac{|C|}{m_x^\xi} \right] |\Phi_\xi(\alpha \theta)|,$$

and therefore

$$|\Phi_\xi^*| \leq |D| \left[ \frac{|C|}{m_x^\xi} \right] |\Phi_\xi(\alpha \theta)| + |d_{0_\xi}| \leq |D| \left[ \frac{|C|}{m_x^\xi} \right] m_\Phi + |d_{0_\xi}|.$$  (3)

Let $m_\Phi$ be the maximum number of $\text{add}$ effects over a single database by a single operator in $D$, defined by

$$m_\Phi \equiv \max_{\alpha \theta \in D} \left( |\Phi_f(\alpha \theta)|, |\Phi_w(\alpha \theta)|, |\Phi_v(\alpha \theta)| \right).$$  (4)

By construction we have that

$$DS(x) \subseteq 2^{\Phi_f^*} \times 2^{\Phi_w^*} \times 2^{\Phi_v^*} \subseteq DS.$$

Therefore, using (3), we prove the theorem:
\[ |DS(x)| \leq |2^{\phi_f^*} \times 2^{\phi_w^*} \times 2^{\phi_v^*}| \leq 2^{(|\phi_f^*| + |\phi_w^*| + |\phi_v^*|)}, \]
\[ |\phi_f^*| + |\phi_w^*| + |\phi_v^*| \leq |D| \left[ \frac{|C|}{m_{\phi}} \right] m_{\phi} (|d_0| + |d_{0_w}| + |d_{0_v}|). \]
\[ |DS(x)| \leq 2^{\left[D\left[ \frac{|C|}{m_{\phi}} \right] m_{\phi} \parallel d_0 \parallel\right] \|d_0\|} \]

with \( \|d_0\| \) the total number of formulas present in the three databases at the initial state defined in Theorem 17. \( \Box \)

We prove in Theorem 17 that, for any finite instance \( x \) of the PKS planning problem \( DS(x) \), its reachable database-state space is finite. The following corollary follows trivially:

**Corollary 18.** Given a finite instance \( x = (D, d_0, G) \in I_{PKS} \) of the PKS planning problem any conditional plan \( P \) such that \( P \in SOL_{PKS} \), i.e., that is a solution of \( x \), has a finite number of nodes bounded by the cardinality of \( DS(x) \), i.e.:
\[ |V(P)| \leq |DS(x)|. \]

**Proof.** The relation \( \leq \) obviously holds by the definition of \( DS(x) \). A conditional plan \( P \) that is a solution of \( x \), irrespective of being \( P \) a strong, strong cyclic, weak and weak cyclic solution can never visit database-states outside \( DS(x) \). \( \Box \)

### 3.2. The PKS knowledge-state space

Given any instance \( x \in I_{PKS} \) of the PKS planning problem, we define the set \( KS(x) \subseteq KS \), called the **reachable knowledge-state space for** \( x \), by
\[ KS(x) \equiv \{ k \mid \exists d \in DS(x) \wedge w = T_{MLK}(d) \wedge k = [w]_m \}. \]

**Corollary 19.** Given a finite instance \( x \in I_{PKS} \) of the PKS planning problem their corresponding reachable knowledge-state space \( DS(x) \) is finite and bounded by the number of database-states of the reachable database-state space:
\[ |KS(x)| \leq |DS(x)|. \]

**Proof.** The \( \leq \) relation obviously holds by the definition of \( KS(x) \), and we know that \( DS(x) \) is finite by Theorem 17. \( \Box \)

### 3.3. The PKS reasoning capabilities

**Remark 20.** The reasoning capabilities of PKS for planning are only based on the database-states.

Definition 9 of an instance of the PKS planning problem and Definition 16 of its solutions are based only in the database-state space \( DS \). Clearly, through the function \( T_{MLK} \)
and the MLK-entailment equivalence relation (see Definitions 2 and 4) we can always correlate any database-state with a certain knowledge-state, but only the database-states are used and no reference to the knowledge-states is needed in this section and Section 2.4 that formally defines an instance \( x \in I_{PKS} \) of the planning problem and its solutions \( SOL_{PKS}(x) \).

The present formalization has clarified how PKS reasons when planning; this has been achieved by defining the instances of the PKS problem and its solution only based on \( DS \).

Even if the concept of database-state (distinct from the knowledge-state) was implicitly present in [3,24], we show in addition that the reasoning mechanism of PKS is based exclusively on the database-states (\( DS \)), and not on the knowledge-states themselves (\( KS \)). This clarification was the key to construct the complete planning algorithm given in Section 3.5.

### 3.4. The PKS complexity

The result expressed in Theorem 22 is a direct product of the presence of the parameters in the operators. The factor \( O(|C|!) \) in the size of \( d \) comes from the need of considering all the possible ways of instantiating the operators. This can be better understood by seeing the associated result expressed in Theorem 21.

We define \( PKS^{-i} \) as the subset of the PKS planning problem in which the operators are restricted to be parameterless. \( I_{PKS^{-i}} \) denotes the subset of instances of the \( PKS^{-i} \) planning problem. Therefore:

\[
I_{PKS^{-i}} \equiv \{ x \mid x = (D, d_0, G) \in I_{PKS} \land \forall \alpha \in D.\forall \gamma(\alpha) = \emptyset \} \subset I_{PKS}.
\]

**Theorem 21.** The plan existence problem in \( PKS^{-i} \) is in PSPACE.

**Proof.** Because PSPACE = NPSPACE it suffices to present a nondeterministic algorithm that uses at most polynomial space and solves the problem to prove that PKS is in PSPACE. In Section 3.5 we describe PlanPKS* a nondeterministic algorithm that solves the problem. It is easy to show that this algorithm uses only a space that is a polynomial on \( x \) to save the current state \( d \). In the worst case, we may have in the databases whose contents represent \( d \) all the formulas that can be added by all the parameterless operators in \( D \) plus the formulas present in the database at the initial state \( d_0 \), and clearly both: \( \|d_0\|, |D| \in O(|x|) \).

**Theorem 22.** The plan existence problem in PKS is at least in \( EXP \).

**Proof.** It is enough to prove that PKS is not in PSPACE = NPSPACE. Any nondeterministic algorithm needs to use space to save the current state \( d \) and the size of \( d \) is not polynomial on the size of \( x \) in the worst case. We may need to add to the databases all the formulas that can be added by all the instantiated operators in \( ED(x) \), and \( |ED(x)| \in O(|D||C|!) \) (see Eq. (2) in Section 3.1) is not polynomial in the size of \( x \) because of the factorial \( |C|! \). Therefore, any nondeterministic algorithm may need to use more than polynomial space to represent \( d \).
3.5. PlanPKS*: a sound and complete planning algorithm for PKS

PlanPKS is the planning algorithm for PKS published in [24]. When it expands a plan under construction to possibly new knowledge-states, it does not check if those knowledge-states are already present in the plan. This means that the algorithm fails to identify possible cycles when searching for a plan. In other words, PlanPKS constructs a tree and not a graph. If the corresponding graph has cycles, PlanPKS attempts to generate an infinite tree. Therefore, it may never terminate. There is a short comment in [24] that states “some forms of cycle checking” are used, but no information is given in the paper about how that cycle checking is done.

Identifying if two different strings in $L_{MLK}$ represent the same knowledge-states in $KS$ implies proving that both states logically entail the same set of formulas, and thus, they are in the same MLK-entailment equivalence class. Due to the expressiveness of $L_{MLK}$ (see Definition 4, and notice that it contains universal quantification), this is certainly impossible for any practical application as was already noted in [3] (see its Section 3, which refers to [19]).

This is not the case for identifying two database-states as the same. If each database-state is written using a certain unique order, e.g., alphabetical order of all the formulas that are the contents of each of the components of $d = (d_f, d_w, d_v)$, we can test the equivalence of two database-states in $O(||d||)$ time. Clearly, it is possible to improve this algorithm using balanced binary-trees, similar structures to keep the order, or some kind of hashing, to identify each database-state. Independently of how this test is done, we now introduce PlanPKS*, a nondeterministic planning algorithm that checks for equivalent database-states when searching for a plan, thus identifying repeated states in $P$. PlanPKS* is shown in Algorithm 1.

Having four kinds of solutions, strong, strong cyclic, weak acyclic and weak cyclic, we have four PKS planning problems, and therefore, we need four planning algorithms to search for those four kinds of solutions. It is more convenient, however, to enclose in one algorithm these four algorithms by designing a unique PlanPKS* whose behaviour changes according to one of its parameters. PlanPKS* has two input parameters: $x \in I_{PKS}$ the instance of the PKS planning problem to be solved and SOLClass $\in \{+, -, +c, -c\}$ used to tailor the behaviour of the algorithm to search for solutions of one specific class. PlanPKS* will search for strong, strong cyclic, weak acyclic or weak cyclic solutions when the value of SOLClass is $+$, $-$, $+c$, $-c$ respectively. PlanPKS* returns an ordered pair whose first element is a Boolean value that states if the algorithm succeeded in finding the requested solution, and whose second element is the found plan (line 26). We use some local variables: $P$ to keep the plan under construction, $L$ a set with all non-expanded nodes of $P$, and others with more obvious purposes. We use three functions: IsDeadEndCycle($P, d, d'$), AddCycle($P, d, d'$) and UpdatePlan($P, L, d, d'$) to hide the standard manipulation of the conditional plan nodes and edges. IsDeadEndCycle($P, d, d'$) return $\top$ when adding an edge from $d$ to $d'$ will close a cycle that is a dead-end cycle, and $\bot$ otherwise. AddCycle($P, d, d'$) adds an edge to $P$ from $d$ to $d'$ in $V(P)$. UpdatePlan($P, L, d, d'$) updates $P$ by adding a new node $d'$, a new edge from $d$ to $d'$ and finally by adding $d'$ to $L$. There is also the function IsSolution($P$, SOLClass) which decides when a certain plan $P$ belongs to the requested class of solutions.
\[ P = (\{d_0(x)\}, \emptyset) \] where
\[ L = \{d_0\} \] //initialize

\[ \text{while} \ (L \neq \emptyset \text{ and } \neg \text{IsSolution}(P, \text{SOLClass})) \text{ do} \]

\[ d = \text{choose}(L) ; \] //nondet. choose current node

\[ \text{split} = \text{choose}(\{\bot, \top\}) ; \] //nondet. choose split or execute

\[ \text{if split} \]

\[ \phi = \text{choose}(d_w) ; \] //with \( d = (d_f, d_w, d_v) \)

\[ \text{ExpandedDatabaseStates} = \{\text{Result}^+(\phi, d), \text{Result}^-(\phi, d)\} ; \]

\[ \text{else} \]

\[ \alpha\theta = \text{choose}(\mathcal{ED}(x)) ; \]

\[ \text{ExpandedDatabaseStates} = \{\text{Result}(\alpha\theta, d)\} ; \]

\[ \text{end if} \]

\[ \text{for all } d' \text{ in ExpandedDatabaseStates do} \]

\[ \text{if } d' \in V(P) \text{ then} \]

\[ \text{if SOLClass} = +\text{ or SOLClass} = - \text{ then} \]

\[ \text{fail} ; \]

\[ \text{else if SOLClass} = +c \text{ and IsDeadEndCycle}(P, d, d') \text{ then} \]

\[ \text{fail} ; \]

\[ \text{else} \]

\[ \text{AddCycle}(P, d, d') ; \]

\[ \text{end if} \]

\[ \text{else} \]

\[ \text{UpdatePlan}(P, L, d, d') ; \]

\[ \text{end if} \]

\[ \text{end for} \]

\[ \text{end while} \]

\[ \text{return} (\text{IsSolution}(P, \text{SOLClass}), \text{Normalize}(P)) ; \]

Algorithm 1. PlanPKS*(x, SOLClass).

As usual, to abstract the different possible search strategies we use the nondeterministic \text{choose}(A), which nondeterministically picks an element of the set \( A \), and \text{fail} which triggers the backtracking. \text{choose}(A) is used in PlanPKS* to nondeterministically choose: first, the node to expand which is saved in \( d \) (line 3); second, if one of the formulas in \( d_w \) should be split or an instantiated operator should be executed (line 4); three, to choose which formula to split (line 6); and lastly, to choose which instantiated operator to execute (line 9). PlanPKS* checks for the previous presence in \( P \) of every new expanded database-state in line 13.

The differences between the four algorithms enclosed in PlanPKS* emerge only in two small parts: They differ in what is done when the expanded node is already present in \( P \) (lines 14–20) and within the function IsSolution. When searching for strong or weak acyclic solutions, the cycles are banned altogether, hence, the backtracking mechanism is invoked by \text{fail} (line 15). When searching for strong cyclic solutions the dead-end cycles are also banned. This is checked by calling the function IsDeadEndCycle(\( P, d, d' \)) (line 16). If that is the case, \text{fail} is invoked. Otherwise, the cycle is accepted and therefore the AddCycle(\( P, d, d' \)) is called (line 19). The function IsSolution(\( P, \text{SOLClass} \)) returns \( \top \) when a certain plan \( P \) belongs to the \text{SOLClass} class of solutions and otherwise returns \( \bot \). The behaviour of this function that determines the successful loop termination (line 2) varies greatly, depending on the \text{SOLClass} passed. When searching for strong or
strong cyclic solutions \( \text{IsSolution}(P, \ldots) \) requires that the goal \( G \) holds in all the terminals of \( P \). When searching for weak acyclic or weak cyclic solutions the function requires that the goal holds at least in one of the terminals of \( P \); but in practice it is necessary to add another criterion to continue searching for weak solutions of better quality and, therefore, the search would be terminated only when a certain quality is reached or by time out.

Finally, we have the function \( \text{Normalize}(P) \) (line 26) that returns a normalized \( P \) trimmed of every dead-end loop and of every unnecessary isolated step to a non-goal terminal. When called over strong solutions, this function does nothing because every terminal satisfy the goal. \( \text{Normalize}(P) \) constructs the normalized \( P \) in the following way. Every dead-end loop is eliminated from \( P \) leaving in its place only the non-goal terminal state that was the entry point to the loop. Every isolated path of more than one step to a non-goal terminal is replaced by its first step. For example, if \( P \) is the solution shown in Fig. 2(d) we have that \( \text{Normalize}(P) \) will trim its unique dead-end loop (by removing nodes numbered 1 and 2 in the figure and the corresponding three steps in the cycle). It will also trim the final unnecessary step that goes to the unique non-goal terminal state (by removing node numbered 3 in the figure, and the step that leads to it).

In the original PKS presentation [24] only the strong solution case was considered: the plans constructed are trees, hence acyclic, and a plan is considered a solution when \( G \) holds in all its terminals. Unlike \( \text{PlanPKS}, \text{PlanPKS}^* \) checks for equivalent database-states at any expansion when searching for the plan (line 13). Even if this increases the computational cost it actually brings an advantage, because it guarantees termination. Recall that Theorem 17 shows that the reachable database-state space \( DS(x) \) associated with any instance \( x \) of PKS is finite, and therefore, if the planning algorithm does not loop in \( DS \) its termination is guaranteed. Hence, \( \text{PlanPKS}^* \) is not only sound (as \( \text{PlanPKS} \)), but also complete. \( \text{PlanPKS}^* \) is a direct product of the formal semantics for PKS introduced in this paper. We could build a sound and complete algorithm only because the clear distinction between database-states and knowledge-states in the present complete formalization of PKS, allowed us to avoid cycles in its reasoning mechanism and hence, guarantees that it always terminates.

In Remark 6 we stated that different database-states can be mapped into the same knowledge-state. Therefore, even if \( \text{PlanPKS}^* \) always identifies cycles in \( DS \) when constructing a plan, it is possible that the associated reasoning in \( KS \) cycles anyway, i.e., we can visit different database-states that are all mapped on the same knowledge-states. At worst this leaves room for improving the efficiency of the planning algorithm, but does not imply that the PKS reasoning mechanism is cycling and may not terminate, because it is based only on the database-states and it is not cycling in \( DS \).

4. A quantitative analysis of plan quality

In this section we deal with the problem of comparing and measuring the quality of solutions that belong to any of the different classes of solutions introduced in Section 2.5: strong [acyclic], strong cyclic, weak acyclic and weak cyclic. We start by showing how intuition can lead us to wrong assumptions.
4.1. Are the four classes of solutions a guide for their quality?

An ever present question in planning concerns the quality of the plans found. When it is possible to find more than one plan, we need to answer the following question: Which solution is better? This simple question is the underlying motivation of this section. But, initially we focus on a less ambitious question: Do the four classes of solutions introduced before give us a real indication of their quality? It seems intuitive that a strong solution is better than a weak one, and that an acyclic property is preferred over a cyclic one. Therefore, when trying to resolve a PKS instance, intuition leads us to first search for a strong solution because it will have both positive characteristics: that is to say, it will be strong and acyclic. If no strong solution is found, we can search for a strong cyclic or weak acyclic solution. If we find only one we may use that solution because it seems that a weak cyclic solution will not be better. On the other hand, if we find both a strong cyclic and a weak acyclic solution, we need to choose which one to use. It is not evident which is better, because both have one of the desired properties and a negative one. If no solution is found, we can still hope to find a weak cyclic solution, which would seem to be the worst case. But, even if intuition can lead us to this kind of reasoning and therefore, to think that the four classes of solutions are a guide for their quality, we will now show that this is not always the case.

The authors of [7] state that “the strong solutions to a planning problem are a subset of the strong cyclic solutions, which are in turn a subset of the weak solutions.” Denoting the set of all weak solutions by $SOL_{PKS}^-(x) = SOL_{PKS}^-(x) \cup SOL_{PKS}^c(x)$ we can write this statement formally as follows:

$$\forall x \in I_{PKS}, \{SOL_{PKS}^+_{PKS}(x) \subseteq SOL_{PKS}^+_{c}(x) \subseteq SOL_{PKS}^-_{PKS}(x) \}.$$ 

In [7] it is stated that “weak and strong solutions correspond to the two extreme requirements for satisfying reachability goals”, and that in cases where “weak solutions are not acceptable and strong solutions do not exist […] strong cyclic solutions are a viable alternative”. These statements were made according to the assumption that in some applications, e.g., safety critical ones, one cannot accept weak solutions of any type. Clearly, by assuming \textit{a priori} that weak solutions are forbidden, we leave them out, and deal only with strong and strong cyclic ones. But this assumption is not useful for every planning problem, hence, we need to consider also weak solutions when answering the question about the quality of plans in general. We show in this section that we cannot take these statements as valid in general, i.e., strong cyclic solutions are not always better than weak ones.

Thanks to the distinction between weak acyclic and weak cyclic solutions we can identify $SOL_{PKS}^+(x)$ as the subset of $SOL_{PKS}^-_{c}(x)$ whose solutions are weak but not cyclic, and therefore do not belong to $SOL_{PKS}^+_{c}(x)$. In particular, it is not difficult to see from Definition 16 that only the following inclusion relations hold:

$$\forall x \in I_{PKS}, \{SOL_{PKS}^+_{PKS}(x) \subseteq SOL_{PKS}^+_{c}(x) \subseteq SOL_{PKS}^-_{PKS}(x) \}.$$
The fact that $SOL_{PKS}^-(x)$ and $SOL_{PKS}^c(x)$ are not related by inclusion should highlight that it is not right to assume that in any case a strong cyclic solution is preferable to weak acyclic ones. Indeed, this assumption is always valid in the limit case of infinite execution, i.e., if we accept to wait forever for the execution of the plan to terminate. But what is the point of a quality consideration that only holds in the infinite execution case? Hence, it is necessary to pay attention to what happens in more realistic situations where plans are expected to finish after a certain number of steps.

We prove in the rest of this section, by means of a simple example, that:

**Remark 23.** If we do not assume the execution of a plan to run forever:

1. There are weak acyclic plans that are better than strong cyclic ones.
2. The classification of plans in strong, strong cyclic, weak acyclic and weak cyclic does not give us any indication of their quality as solutions.

Since plans in PKS are represented as cyclic directed graphs, it is clear that the number of steps a plan $P$ executes depends not only on the number of its nodes $|E|$ and edges $|V(P)|$, but also on the morphology of the graph. For example the execution of an acyclic graph will necessarily end after at most $|V(P)|$ steps, but it can run forever if $P$ is cyclic. Therefore, to compare the quality of different solutions, we will consider the probability of reaching a database-state which satisfies the goal condition at $k$ steps of running a given conditional plan $P$. We call it the **probability of success at $k$ steps for $P$**, denoted by $P^G_k(P)$. The example that follows shows that this probability can be greater for some weak acyclic solutions than for strong cyclic ones, and therefore, that strong cyclic solutions are not always to be preferred over weak acyclic ones.

Fig. 3 shows two solutions of a certain instance $x \in I_{PKS}$. $P^a$ is a weak acyclic solution, i.e., $P^a \in SOL_{PKS}^-(x)$, and $P^b$ is a strong cyclic solution, i.e., $P^b \in SOL_{PKS}^c(x)$. $G$ denotes the number of terminals of the solution in which the goal condition holds; $F$

---

**Fig. 3.** A weak acyclic solution that is better than a strong cyclic solution.
Table 3
The probability of success after $k$ steps for $P_a$ and $P_b$

<table>
<thead>
<tr>
<th>$k$</th>
<th>$P^k_G(P_a)$</th>
<th>$P^k_G(P_b)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k \leq 1000$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$1000 &lt; k \leq 2^{1000}$</td>
<td>$\sim 1$</td>
<td>$0$</td>
</tr>
<tr>
<td>$2 \times 2^{1000} &lt; k \leq 3 \times 2^{1000}$</td>
<td>$\sim 1$</td>
<td>$\frac{1}{2}$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>$n \times 2^{1000} &lt; k \leq (n+1) \times 2^{1000}$</td>
<td>$\sim 1$</td>
<td>$\frac{1}{n+1}$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>$2^{1000} \times 2^{1000} &lt; k \leq (2^{1000} + 1) \times 2^{1000}$</td>
<td>$\sim 1$</td>
<td>$= 1$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$\ldots$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>$k \to \infty$, $n \to \infty$</td>
<td>$\sim 1$</td>
<td>$1$</td>
</tr>
</tbody>
</table>

denotes the number of terminals of the solution in which the goal does not hold; and $C$ denotes the number of database-states that are visited by the cyclic path in the strong cyclic solution $P_b$.

In general, the value of $P^k_G(P)$ changes depending on both $k$ and $P$. We assume that for every disjunction both paths are equally probable. For the weak acyclic conditional plan $P_a$ if $k \leq 1000$ the conditional plan $P_a$ cannot possibly reach a goal database-state, and therefore, $P^k_G(P_a) = 0$. If $1000 < k$ the terminal states of $P_a$ have been reached, and therefore, its probability of success is

$$P^k_G(P_a) = \frac{G}{G + F} = \frac{2^{1000}}{2^{1000} + 1} \sim 1.$$ 

Instead, for the strong cyclic solution $P_b$, the probability of success after at most $k$ steps is given by $P^k_G(P_b) = \frac{n}{n+1}$, for $k$ in the range $n \times C < k \leq (n + 1) \times C$ with $n = 0, 1, 2, \ldots, \infty$. Notice that the strong cyclic solution $P_b$ cannot possibly reach a goal state before $C = 2^{1000}$ steps, and therefore $P^k_G(P_b) = 0$ for $k \leq 2^{1000}$. Table 3 shows the values of $P^k_G$ for the different values of $k$ for both solutions. It is easy to see that $P_a$ has a larger probability of success than $P_b$ for all values of $k$ greater than 1000 and until $k = 2^{1000} \times 2^{1000} + 1$ when we have that both probabilities are equal

$$P^k_G(P_a) = P^k_G(P_b) = \frac{2^{1000}}{2^{1000} + 1} \sim 1,$$

and that only in the limit it is one

$$\lim_{k \to \infty} P^k_G(P_b) = \lim_{n \to \infty} \frac{n}{n + 1} = 1.$$

The previous example proves that, contrary to intuition, a strong cyclic solution is not always better than a weak acyclic one. But obviously, neither the contrary is true. Often a strong cyclic solution is better than an acyclic one. For example a weak acyclic solution $P_a'$ like $P_b$ but in which the values of $G$ and $F$ are swapped, with a strong cyclic solution $P_b'$ like $P_b$ but with $G = C = 2^{1000}$. For most values of $k$ the first will have $P^k_G(P_a') \sim 0$ meanwhile the second will be $P^k_G(P_b') \sim \frac{k}{k+1} \sim 1$. 
Using similar reasoning, it is not difficult to prove that when dealing with solutions at a certain number of steps $k$, certain strong solutions can be worse than weak ones, even weak cyclic solutions. It suffices to consider a strong solution large enough to guarantee that it does not finish its execution before $k$ steps to see that it is possible for a weak solution to have a larger probability of success after those steps.

So, we cannot rely on the different classes of solutions as a quality indicator. In consequence, the question we posed at the beginning of this section, about which solutions are better, has become more important. We have revealed the lack of existence of a method to analyze the quality of solutions. In what follows we present one. In the next section, we present quantitative quality measures that can be applied to various kinds of solutions. In the subsequent section, we show how to use these measures to compare solutions and choose the best ones. In particular, we provide a quantitative method for choosing the best solution between two solutions from any class.

### 4.2. A quantitative measure of plan quality for comparing solutions

In the previous section, we highlighted the idea that in most cases the quality of a solution depends on the number of steps we consider important for the problem. We postulate here the probability of success at $k$ steps for any conditional plan $P$ as a quantitative general measure of its quality at that point of execution, independent of which class of solution $P$ is in, i.e., independent of it being strong, strong cyclic, weak acyclic or weak cyclic.

**Definition 24.** Given any instance $x \in I_{PKS}$ of the PKS planning problem and any of its solutions $P \in SOL_{PKS}(x)$, we define $q(k, P)$, the quality measure of $P$ for $k$ steps of execution, as the probability of success for $P$ at $k$ steps of execution, formally $q(k, P) = P_k^G(P)$.

This quality measure allows us to determine exactly which plan is preferable at a certain number $k$ of steps executed regardless of which class the plan belongs to.

**Remark 25.** A solutions comparison method for $k$ steps of execution for any two solutions $P^a \in SOL_{PKS}(x)$ and $P^b \in SOL_{PKS}(x)$ of the same instance $x \in I_{PKS}$ is given by:

1. $q(k, P^a) < q(k, P^b) \Rightarrow P^b$ is preferable for $k$ steps of execution;
2. $q(k, P^a) = q(k, P^b) \Rightarrow$ both are equivalent for $k$ steps of execution;
3. $q(k, P^a) > q(k, P^b) \Rightarrow P^a$ is preferable for $k$ steps of execution.

In Section 4.5, we introduce a general comparison method for solutions of any class that is also based on the quality measure $q(k, P)$. Hence, $q(k, P)$ is extremely useful. But alas, it is not easy to compute $q(k, P)$, because it depends on many factors and it changes from problem to problem.

Surprisingly, if we assume that for every disjunction in $P$ both paths are equally probable, it is easy to compute the probability of success $q(k, P)$ for any acyclic conditional plans $P$. In the next section we show $q^a(k, P)$ a polynomial (actually linear) algorithm
that makes this assumption to compute the probability of success $q(k, P)$. We propose to use $q^a(k, P)$ for approximating $q(k, P)$.

Sadly, this is not the case for cyclic solutions. In these cases, even if it is not difficult to extend the algorithm we have shown for acyclic solutions to compute $q(k, P)$ for cyclic solutions, this algorithm is of much greater computational complexity, and hence, not of much use. We have preferred to provide here $q^c(k, P)$, a more efficient but less precise approximation of $q(k, P)$.

When dealing with acyclic solutions we can wait until they finish their execution. Therefore, it is useful to provide an absolute measure of their quality that assumes that $k$ is big enough that we can be sure the $P$ terminated. Clearly, for any acyclic solution choosing $k = |V(P)|$ is enough.

**Definition 26.** Given any instance $x \in I_{PKS}$ of the PKS planning problem and any of its acyclic solutions $P \in SOL^a_{PKS}(x)$, we define the absolute quality measure of the acyclic solution $P$, denoted by $Q(P)$, as

$$Q(P) \equiv q\left(|V(P)|, P\right).$$

For every strong solution $P \in SOL^+_{PKS}(x)$ of a PKS instance $x$, we always have $Q(P) = 1$. Vice versa, $Q(P) = 1$ states that $P$ is a strong solution. It is clear that $Q(P)$ is useless as a quality measure for comparing strong solutions, because it only measures the quality of acyclic solutions after they have terminated their execution, and by definition all strong solutions guarantee success if we wait for their termination.

Conversely, for weak acyclic solutions $P \in SOL^-_{PKS}(x)$ of a PKS instance $x$, we always have $Q(P) < 1$. Hence, if we can wait until the end of the execution of the acyclic solutions, $Q(P)$ correctly tells us to always prefer strong solutions over weak ones. In general, values of $Q(P) \sim 1$ mean that the weak acyclic solution $P$ is *almost a strong solution* because it has almost as many terminals that satisfy the goal condition as the total number of terminals. Values of $Q(P) \sim 0$ mean that $P$ is *almost not a solution* because almost none of its terminals satisfy the goal condition.

It is important not to misunderstand in which sense $Q(P)$ gives an *absolute measure* of the quality of an acyclic plan $P$. This measure is *absolute* in the sense that it does not depend on the number $k$ of steps as $q(k, P)$ does, but we are assuming $k$ big enough to ensure $P$ has terminated its execution. Therefore, we need to be careful when comparing the quality of different acyclic plans using $Q(P)$. In those cases, it is necessary to keep in mind that we are comparing their quality after *every plan has finished its execution*. Hence, in case we have $Q(P_1) > Q(P_2)$ it is wrong to assume that $P_1$ is better for every number $k$ of execution steps, because the quality measure is valid only after both plans have terminated. Clearly, $P_2$ might terminate much earlier than $P_1$ and therefore, for smaller values of $k$ $P_2$ can be better than $P_1$.

We have presented here the quality measure $q(k, P)$ and a comparison method valid at a certain step $k$ of execution that is based on that quality measure. We continue by showing how to approximate $q(k, P)$ in practice.
SuccessProb = 0
LDFT(k, P)
LDFT.CurrentNode.Prob = 1
while LDFT.NextNode() do
  if LDFT.PreviousStepIsSplit() then
    LDFT.CurrentNode.Prob = 1/2 × LDFT.PreviousNode.Prob
  end if
  if η(G, LDFT.CurrentNode) = ⊤ then
    SuccessProb = SuccessProb + LDFT.CurrentNode.Prob
  end if
end while
return SuccessProb

Algorithm 2. q^a(k, P).

4.3. \( q^a(k, P) \): approximating \( q(k, P) \) for acyclic solutions

We now introduce \( q^a(k, P) \), a quality measure that approximates \( q(k, P) \) for acyclic solutions.

**Definition 27.** Given any instance \( x \in I_{PKS} \) of the PKS planning problem and any of its acyclic solutions \( P \in SOL_{PKS}(x) \), we define \( q^a(k, P) \), the quality measure of an acyclic solution \( P \) in at most \( k \) steps, by means of Algorithm 2.

The algorithm \( q^a(k, P) \) approximates the probability of success at \( k \) steps of execution \( P^k_{\sigma}(P) \) for any acyclic solution assuming that both paths of every disjunction in \( P \) are equally probable. Therefore, when this assumption holds \( q^a(k, P) \) computes \( q(k, P) \) exactly, otherwise, its value as an approximation degrades.

To simplify the writing of this algorithm we have assumed the existence of an object, referred to as \( LDFT \), that implements the capabilities of traveling a conditional plan in a limited depth-first mode, plus some features we explain here. \( LDFT \) helps us to abstract all the standard parts of the algorithm (the traveling through the plan nodes) to highlight the specific part of it (the calculus of \( q(k, P) \)). This object is constructed by calling \( LDFT(k, P) \), when \( k \) is the limit of depth and \( P \) is the plan \( LDFT \) will travel. When initializing, \( LDFT \) sets its member variable \( LDFT.CurrentNode \) to the root of \( P \). This variable will always point to the current node. Its member function \( LDFT.NextNode() \) returns \( \top \) or \( \bot \) depending on whether of not it succeeds in advancing to a next non-visited node of \( P \) in a depth-first mode. \( LDFT.NextNode() \) does not succeed when the traveling has finished, and therefore the algorithm stops looping. If the execution splits a database-state, it travels first to the left node and after backtracking, to the other node. We have followed a left-first traveling approach, but equivalently we could have taken a right-first one. \( LDFT.NextNode() \) stops advancing through a path when it has already reached a terminal goal or because it has already reached the depth of \( k \) steps. In this cases \( LDFT.NextNode() \) will backtrack to the nearest previous splitting node at which there is a path that has not been explored and, after that, to the first node of that non-explored path, setting \( LDFT.CurrentNode \) properly to that node. After calling the member function \( LDFT.NextNode() \) we can use \( LDFT.CurrentNode \) to access the current node. We assume
that we can save a real number in each node of the conditional plan $P$ with the probability of reaching that point, i.e., in the variable: $LDFT.CurrentNode.P$. We use two more members: The function $LDFT.PreviousStepIsSplit()$ that returns $\top$ if the step that has finished in the $LDFT.CurrentNode$ has been a split of a database-state and $\bot$ otherwise; and $LDFT.PreviousNode$ that gives access to the previous node.

In Fig. 4 we have placed the traveling sequence just above each node. In the left part of the figure the numbers below each node show the probability of reaching that node. In the right part the numbers show the actual value of $SuccessProb$. The dotted line represents the limit at which $LDFT$ stops traveling when $qa(k, P)$ is called with $k = 4$. It stops before the 10th node and hence the final $SuccessProb$ is $1/2$. Instead, for larger values of $k$ it reaches the 10th node and the final value of $SuccessProb$ is $3/4$.

As said in the previous section, when dealing with acyclic solutions we can wait until they finish their execution, and therefore, it is useful to use the absolute quality measure $Q(P)$ defined in terms of $q(k, P)$ by using a $k$ big enough to be sure the $P$ has terminated. Hence, it is natural to introduce $Q_a(k, P)$, an approximation of $Q(P)$ that is defined in terms of $qa(k, P)$, in the same way as $Q(P)$ is defined in terms of $q(k, P)$.

**Definition 28.** Given any instance $x \in I_{PKS}$ of the PKS planning problem and any of its acyclic solutions $P \in SOL_{PKS}(x)$, we define the approximation of the absolute quality measure of the acyclic solution $P$, denoted by $Q_a(P)$, as

$$Q_a(P) \equiv qa(|V(P)|, P).$$

For example, the conditional plan $P^a$ shown in Fig. 3(a) is almost a strong solution because its quality measure is $Q_a(P^a) \sim 1$, and notice that this is exactly the value of $P^a_k$ for $k > 1000$.

**4.4. $q^c(k, P)$: approximating $q(k, P)$ for cyclic solutions**

We now introduce $q^c(k, P)$ a quality measure that approximates $q(k, P)$ for cyclic solutions. To simplify the exposition, we first give the definition of $q^+c(k, P)$ that approximates $q(k, P)$ only for strong cyclic solutions and after that, we use it to introduce $q^c(k, P)$ for any cyclic solutions. Again, we assume that both paths of every disjunction in $P$ are equally probable. But contrary to the case of $q_a(k, P)$, even when this assumption holds, neither $q^+c(k, P)$ nor $q^c(k, P)$ compute $q(k, P)$ exactly. As we said, even if it is not difficult to
deliver an algorithm to compute \( q(k, P) \) exactly under this assumption, it is not of much use due to its high computational complexity. Hence, we have preferred to deliver a more efficient if less precise approximation.

**Definition 29.** Given any instance \( x \in I_{PKS} \) of the PKS planning problem and any of its strong cyclic solutions \( P \in SOL_{PKS}^{+c}(x) \), we define the **quality measure of a strong cyclic solution \( P \) in at most \( k \) steps**, denoted by \( q^{+c}(k, P) \) as follows:

\[
q^{+c}(k, P) = \frac{\text{MFC}(k, P)}{\text{MFC}(k, P) + 1}
\]

being \( \text{MFC}(k, P) \) a measure of the mean number of favorable cases of \( P \) after \( k \) steps, defined by

\[
\text{MFC}(k, P) = \frac{k|T(P)|}{\|\text{LCP}(P)\||\text{MFVS}(P)|}
\]

where

- \( \text{LCP}(P) \) is the Longest Cyclic Path of \( P \).
- \( \text{MFVS}(P) \) is the Minimum Feedback Vertex Set of \( P \).

The **Longest Cyclic Path** of \( P \), denoted by \( \text{LCP}(P) \), is a simple cyclic path in \( P = (V, E) \), i.e., a sequence of distinct vertices \( v_1, \ldots, v_m \) of \( V \) such that for any \( 1 \leq i \leq m - 1 \), \((v_i, v_{i+1}) \in E \) and \( v_1 = v_m \) (see the more general Longest Path problem in [2]). Even if computing the longest path is not in APX-Hard it can be approximated by a \( \mathcal{O}(|V|/\log |V|) \) algorithm [1]. The length of \( \text{LCP}(P) \), denoted by \( \|\text{LCP}(P)\| \), is used as a worst case measure of how long the cycles in \( P \) are compared with its size.

The **Minimum Feedback Vertex Set** of \( P \), denoted by \( \text{MFVS}(P) \), is a subset \( V' \subseteq V \) such that \( V' \) contains at least one vertex from every directed cycle in \( P \) (see [2]). Even if computing \( \text{MFVS}(P) \) is APX-Hard it can be approximated by a \( \mathcal{O}(\log |V| \log \log |V|) \) algorithm [15,26]. \( |\text{MFVS}(P)| \) is used as a measure of how cyclic the conditional plan \( P \) is. It is worth noticing that it is also possible to use the Minimum Feedback Arc Set of \( P \) instead of \( \text{MFVS}(P) \) for the same purpose of measuring how cyclic \( P \) is, but being it also APX-Hard and approximated by a \( \mathcal{O}(\log |V| \log \log |V|) \) algorithm [15], there is no gain in using it.

Notice that \( \|\text{LCP}(P)\| \) and \( |\text{MFVS}(P)| \) are always greater than 0 for strong cyclic solutions, and therefore, \( q^{+c} \) is well defined.

\( q^{+c}(k, P) \) estimates the probability of success of \( P \) after \( k \) steps. Therefore, values of \( q^{+c}(k, P) \sim 1 \) mean that the strong cyclic solution \( P \) is almost a strong solution after executing \( k \) steps. Values of \( q^{+c}(k, P) \sim 0 \) mean the contrary, that \( P \) has a probability of success after \( k \) steps which is close to zero and, therefore, it is almost not a solution for that number of steps. This happens when \( P \) has too few terminals compared with how many cycles it has and with how long these cycles are. For example, for the strong cyclic solution \( P^b \) shown in Fig. 3(b), we have for \( k = C = 2^{1000} \) that \( \text{MFC}(C, P^b) = \frac{C \times G}{2C+1} = G \) and \( q^{+c}(C, P^b) = \frac{G}{2C+1} = \frac{1}{2} \), which is exactly the value of \( P^G_{C}(P^b) \) for \( k = C + 1 \). We have that \( q^{+c}(2C, P^b) = \frac{2}{3} \) is exactly like \( P^G_{C}(P^b) \) for \( k = 2C + 1 \). Both \( q^{+c}(k, P^b) \) and \( P^G_{C}(P^b) \)
coincide for any \( k = rC + 1 \) for \( r = 0, 1, 2, \ldots, \infty \). The equivalence occurs when \( k = C + 1 \), rather than \( k = C \), because \( P^h \) is an extreme case that has only one disjunction which terminates into a terminal state of \( P \) exactly at the end of the cycle. \( q^{+c}(k, P^h) \) estimates \( P^G_k(P^h) \) as if the disjunctions with terminals were uniformly distributed throughout the cycle.

The definition of \( q^{+c}(k, P) \) estimates the probability of success of any \( P \) after \( k \) steps in the following way. The definition of \( q^{+c}(k, P) \) is the expression for the probability of success after \( k \) steps of a conditional plan composed of \( |MFVS(P)| \) cycles, each of length \( LCP(P) \) (worst case), and such that the disjunctions that lead to terminals of \( P \) are uniformly distributed over these cycles. This implies that each cycle has only \( 1/|MFVS(P)| \) of the total number of terminals in \( P \). We show this in Fig. 5. On the left, an example of any strong cyclic solution \( P \). On the right, \( P' \), which is how the solution \( P \) is seen when estimating its probability of success after \( k \) steps. In other words we want a quality measure such that \( q^{+c}(k, P) \) can be approximated as \( P^G_k(P') \) for any strong cyclic solution \( P \), \( P' \) which is how the solution \( P \) is seen when estimating its probability of success after \( k \) steps.

It is enough to notice that \( kC||LCP(P)||/||MFVS(P)|| = MFC(k, P) \) to see that we have explained the definition of \( q^{+c}(k, P) \) (Definition 29). Now we give the definition of \( q^c(k, P) \) that approximates \( q(k, P) \) for both strong and weak cyclic solution.
Definition 30. Given any instance \(x \in I_{PKS}\) of the PKS planning problem and any of its cyclic solutions \(P \in SOL_{PKS}(x)\), we define the quality measure of any cyclic solution \(P\) in at most \(k \) steps, denoted by \(q^c(k, P)\) as follows:

\[
q^c(k, P) \equiv \frac{|T_G(P)|}{|T(P)|} \times q^+c(k, \text{Normalize}(P)),
\]

where

- \(\text{Normalize}(P)\) returns a normalized \(P\) trimmed of every dead-end loop and of every unnecessary isolated step to a non-goal terminal;\(^6\)
- \(T_G(P) \equiv \{d \mid d \in T(P) \land \eta(G, d) = \top\}\) is the set of terminal goals.

It is important to use \(\text{Normalize}(P)\) here to eliminate all dead-end loops before computing \(q^+c(k, P)\). Otherwise the presence of these loops can ruin the approximation \(q^+c(k, P)\) by altering the values of \(\text{LCP}(P)\) or \(\text{MFVS}(P)\). Notice that if \(P\) is a strong cyclic solution \(\text{Normalize}(P) = P\) and \(\frac{|T_G(P)|}{|T(P)|} = 1\), and hence \(q^c(k, P) = q^+c(k, P)\) as expected. But when the solution is weak cyclic, the added factor corrects the value of \(q^+c(k, P)\).

When most terminals do not satisfy the goal, we have \(\frac{|T_G(P)|}{|T(P)|} \sim 0\) and \(q^c(k, P) \sim 0\). When most terminals satisfy the goal, we have \(\frac{|T_G(P)|}{|T(P)|} \sim 1\) and hence \(q^c(k, P) \sim q^+(k, P)\), as expected.

### 4.5. Comparing strong, strong cyclic, weak acyclic and weak cyclic solutions

In this section we describe a general comparison method for solutions of any class based on the quality measure \(q(k, P)\). This method is valid for comparing solutions of any kind, be they strong, strong cyclic, weak acyclic or weak cyclic.

Remark 31 (Solutions comparison method). Given any two solutions \(P^a \in SOL_{PKS}(x)\) and \(P^b \in SOL_{PKS}(x)\) of any class for the same instance \(x \in I_{PKS}\), and being \(k_{\text{max}} \equiv \max(|V(P^a)|, |V(P^b)|)\), we have three cases:

1. \(q(k_{\text{max}}, P^a) \ll q(k_{\text{max}}, P^b)\) ⇒ \(P^b\) is preferable;
2. \(q(k_{\text{max}}, P^a) \sim q(k_{\text{max}}, P^b)\) ⇒ \(P^a\) and \(P^b\) are similar;
3. \(q(k_{\text{max}}, P^a) \gg q(k_{\text{max}}, P^b)\) ⇒ \(P^a\) is preferable.

Both \(q(k_{\text{max}}, P^a)\) and \(q(k_{\text{max}}, P^b)\) quantities are in fact the probability of success after \(k = \max(|V(P^a)|, |V(P^b)|)\) steps of execution and therefore, we can use standard statistical calculus to quantify the probability of being right in choosing \(P^b\) in (1) or in choosing \(P^a\) in (3). In the intermediate case (2) it is possible to estimate more precise values of \(k\) considering the particular instance of the problem and use \(q(k, P)\) for a more adjusted comparison.

\(^6\) \(\text{Normalize}(P)\) was introduced in Section 3.5 when explaining Algorithm 1, which also uses it.
When one (or both) of the plans we want to compare are acyclic we can use $Q(P)$ instead of $q(k, P)$. For example, when given an acyclic solution $P^a \in \text{SOL}^a_{\text{PKS}}(x)$ and a cyclic one $P^b \in \text{SOL}^c_{\text{PKS}}(x)$ for the same instance $x \in I_{\text{PKS}}$, being $k_{\text{max}} \equiv \max(|V(P^a)|, |V(P^b)|)$ we have the following three cases:

1. $Q(P^a) \ll q(k_{\text{max}}, P^b)$ $\Rightarrow$ $P^b$ is preferable,
2. $Q(P^a) \sim q(k_{\text{max}}, P^b)$ $\Rightarrow$ $P^a$ and $P^b$ are similar,
3. $Q(P^a) \gg q(k_{\text{max}}, P^b)$ $\Rightarrow$ $P^a$ is preferable.

In general, when comparing an acyclic solution $P^a$ with a cyclic solution $P^c$ of the same instance, we need to determine first the number $k$ of steps to be used when computing the quality of the cyclic solution: $q(k, P^c)$. We know that being $P^a$ acyclic it will stop after a finite number of steps, but this is not the case for $P^c$. Therefore, to compare these two solutions, we need to determine how many steps we are willing to endure in order for $P^c$ to finish. Clearly, this depends on the problem at hand. This is the reason why we introduced $q(k, P)$, a quality measure that can be adjusted to different values of $k$ adapted for every particular case. In most cases it is useful to have a method for comparing these two kinds of solutions which can deal with cases where one of them is clearly preferable. Notice that the acyclic solution $P^a$ will have at least $\log_2(|V(P^a)|)$ steps (the case of a symmetric binary tree) and at most $|V(P^a)|$ steps (the case of a unique linear sequence). Therefore, the worst case is $k = |V(P^a)|$. Due to the fact that usually cyclic solutions are shorter than acyclic ones, it might be wrongly assumed that it is enough to use this value of $k$ to compute $q(k, P^c)$ and then compare this quality measure with $Q^a(P^a)$. For a comparison method to be valid in general, we also need to include the case of strong cyclic solutions that are bigger than weak acyclic ones against which we want to compare them. Hence, it is necessary to choose the size of the biggest solution: $k = \max(|V(P^a)|, |V(P^c)|)$.

It is worth highlighting that the comparison methods described here is intentionally based on the quality measure $q(k, P)$. This allows us to provide a method that is valid independently of how $q(k, P)$ is computed or approximated. Hence, the method continues to be valid if more efficient or precise approximations of $q(k, P)$ are found in the future, be they general purpose approximations or specifically adjusted for a particular problem.

In practice, we propose to use the approximations given in the previous two sections. Hence, for applying the comparison method we use:

$q(k, P) \sim \begin{cases} q^a(k, P), & \text{if } P \in \text{SOL}^a_{\text{PKS}}(x), \\ q^c(k, P), & \text{if } P \in \text{SOL}^c_{\text{PKS}}(x), \end{cases}$

$Q(P) \sim Q^a(P)$.

The reader should be aware that all the approximations given in the previous two sections assume that both paths of a disjunction are equally probable. When this is not the case, the values obtained by the approximations will not correspond to $q(k, P)$, i.e., with the real probability of success of $P$ at $k$ steps, and consequently, the usefulness of the $q^a(k, P)$ and $q^c(k, P)$ as approximations of $q(k, P)$ degrade. Hence, in these cases, to avoid the comparison method to equally degrade we need to better approximate $q(k, P)$ by taking account of the particular case. The same is valid with respect to $Q(P)$ because it is defined in terms of $q(k, P)$.
In [9] the authors focus on cases in which it is not possible to find a strong cyclic solution. They highlight that when dealing with weak solutions, CTL [13], the language usually used to express extended goals cannot express the distinction between weak solutions that have only one path and solutions that do much better. As a first attempt to overcome this limitation, the authors introduce a new language for extended goals that can express more complex conditions than the existence of a single path and allow them to specify roughly qualitative distinctions. Hence, [9] underlines the importance of this paper that provides for the first time a quantitative method for comparing solutions of any class, independent of them being strong, strong cyclic, weak acyclic or weak cyclic.

5. Conclusions

In this paper we have analyzed, formalized, extended and studied properties of PKS, a framework for planning in domains with incomplete information and sensing. We consider PKS very interesting because its features make it a good candidate to become a reference for comparisons with other frameworks.

Our present formalization of PKS clarified the distinction and interrelation between database-states and knowledge-states. This allowed us to understand two important properties of PKS not accounted for in the original papers by Bacchus and Petrick [3,24]: (a) that the reasoning capabilities of the framework depend only on the database-state space (Remark 20); (b) that some different database-states may be mapped into the same knowledge-state, but not vice versa (Remark 6).

The paper extends the conditional plans (trees in the PKS original presentation [24]) to the more general case of graphs and hence allows them to represent cycles. This allows us to introduce four types of solutions to the PKS planning problem: strong, strong cyclic, weak acyclic and weak cyclic. In this way, we start to link PKS with the Planning as Symbolic Model Checking framework for planning with incomplete information and sensing [7,8].

Furthermore, we introduce PlanPKS*, a sound and complete planning algorithm for the formal semantics of PKS given here. This is an improvement with respect to the original planning algorithm PlanPKS that is sound but not complete.

We prove that \( DS(x) \) the reachable database-state space of any finite instance \( x \) of the PKS planning problem is finite. This result is important because it permits us to prove that PlanPKS* is complete. PlanPKS* checks for equivalent states when searching for a plan to avoid cycling and, because \( DS(x) \) is finite, we can guarantee its termination. We prove also that the reachable knowledge-state space \( KS(x) \) is also finite and bounded by \( |DS(x)| \).

We prove that the complexity of the PKS planning problem is intractable: the PKS plan existence problem is at least in \( \mathcal{E}X\mathcal{P} \). We also prove that this is caused by the presence of the operators’ parameters in PKS, because the associated problem restricted to parameterless operators is in PSPACE.

Finally, we prove that, contrary to current belief, given a PKS problem instance, some weak acyclic solutions have a larger probability of success after executing \( k \) steps than some strong cyclic solutions, except in the limit case of \( k \to \infty \). This means that we can-
not determine the quality of solutions based on the class to which they belong (Remark 23). Therefore, we uncovered a problem: the lack of a method for judging the quality of different solutions. We made an important step in resolving this problem by proposing, for the first time, a quantitative quality measure for solutions of any kind, together with appropriate approximations that compute this measure for any kind of solution. This measure allowed us to specify (Remark 31) a quantitative method for the comparison of solutions of any class, be they strong, strong cyclic, weak acyclic or weak cyclic.

Discussion. The underlying motivation of this article is to compare and understand the relationship between different approaches to the problem of planning with incomplete information and sensing. Interesting cross fertilization is produced by this kind of comparative study, improving simultaneously all these different planning frameworks, not to mention the importance of avoiding the duplication of research efforts. Let us show this cross fertilization by commenting on how these comparisons have already influenced this article.

We have already improved PKS in three directions as a direct result of these comparative studies: first, by extending the conditional plans represented originally as trees to graphs, so as to represent cycles by direct inspiration in EDL [11,12]; second, by introducing into PKS four types of solutions inspired by the works about Planning as Symbolic Model Checking [7,8,10]; third, and most important, by providing a sound and complete algorithm for PKS inspired by the sound and complete planning algorithm presented in EDL [11,12].

When extending PKS to include various classes of solutions we identified a subset of the weak solution class, the weak acyclic solutions, not included in the strong cyclic set of solutions. Contrary to current belief (see [7]), we proved that this set contains solutions with a larger probability of success after executing \( k \) steps than some strong cyclic solutions. Therefore, we realized that the various classes of solutions were not a guide for their quality and that the question about which solutions are of higher quality was indeed open. As a consequence, we confronted the problem of how to characterize the quality of these different kinds of solutions and developed a quantitative measure for the quality of these solutions that allows us to compare them quantitatively, based on the probability of success after \( k \) steps. These results were obtained working within the PKS framework, nonetheless, it is important to highlight that these considerations are valid in general for most frameworks for planning with incomplete information and sensing. Therefore, the quantitative measures given here can be applied not only to PKS, but to other frameworks that tackle these kinds of problems such as EDL [11,12] and Planning as Symbolic Model Checking [7,8,10]. Moreover, we hope that, even if pursued within the PKS framework, most future developments will be applicable in those other frameworks. For example, we are now pursuing the idea of using these quantitative quality measures to develop a heuristic that can direct the search for plans of better quality, useful beyond the PKS framework.

Acknowledgements

I thank CONAE for its support to this research. I am grateful to Luigia Carlucci Aiello and Amedeo Cesta for their time and effort in guiding me through this research. I thank
Giuseppe De Giacomo for his initial comments about how to compare different action theories and Giorgio Ausiello for his patient answers. I also thank the members of the Planning and Scheduling Team (PST) at ISTC-CNR, and my colleges at DIS and CONAE for creating a stimulating environment in which this research has been developed. Finally, I thank the anonymous reviewers for their inspiring remarks.

References